SQL Injection (SQLI)

Article Number: 76 | Rating: 4/5 from 1 votes | Last Updated: Tue, Jul 29, 2014 at 10:13 PM

What is SQL Injection?

SQL injection is a code injection technique, used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker).
OR
In other words, SQL injection is a technique where malicious users can inject SQL commands into an SQL statement, via input fields. Injected SQL commands can alter SQL statement and compromise the security of a web application.

SQL injection (SQLI) is considered one of the top 10 web application vulnerabilities.

Incorrectly filtered escape characters

This form of SQL injection occurs when user input is not filtered for escape characters and is then passed into a SQL statement. This results in the potential manipulation of the statements performed on the database by the end-user of the application.

The following line of code illustrates this vulnerability:

```
statement = "SELECT * FROM users WHERE name ='" + userName + "';"
```

This SQL code is designed to pull up the records of the specified username from its table of users. However, if the "userName" variable is crafted in a specific way by a malicious user, the SQL statement may do more than the code author intended. For example, setting the "userName" variable as:

```
' or '1'='1
```

or using comments to even block the rest of the query (there are three types of SQL comments). All three lines have a space at the end:

```
' or '1'='1' --
' or '1'='1' ({
' or '1'='1' /*
```

renders one of the following SQL statements by the parent language:

```
SELECT * FROM users WHERE name = '' OR '1'='1';
SELECT * FROM users WHERE name = '' OR '1'='1' -- ';
```

If this code were to be used in an authentication procedure then this example could be used to force the selection of a valid username because the evaluation of '1'='1' is always true.

The following value of "userName" in the statement below would cause the deletion of the "users" table as well as the selection of all data from the "userinfo" table (in essence revealing the information of every user), using an API that allows multiple statements:

```
a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't
```

This input renders the final SQL statement as follows and specified:

```
SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't';
```

Note: While most SQL server implementations allow multiple statements to be executed with one call in this way, some SQL APIs such as PHP's mysql_query(); function do not allow this for security reasons. This prevents attackers from injecting entirely separate queries, but doesn't stop them from modifying queries.

Incorrect type handling

This form of SQL injection occurs when a user-supplied field is not strongly typed or is not checked for type constraints. This could take place when a numeric field is to be used in a SQL statement, but the programmer makes no checks to validate that the user supplied input is numeric. For example:

```
statement = "SELECT * FROM userinfo WHERE id =" + a_variable + ";"
```

It is clear from this statement that the author intended a_variable to be a number correlating to the "id" field. However, if it is in fact a string then the end-user may manipulate the statement as they choose, thereby bypassing the need for escape characters. For example, setting a_variable to

```
1;DROP TABLE users
```

will drop (delete) the "users" table from the database, since the SQL becomes:

```
SELECT * FROM userinfo WHERE id=1;DROP TABLE users;
```

Blind SQL injection

Blind SQL Injection is used when a web application is vulnerable to an SQL injection but the results of the injection are not visible to the attacker. The page with the vulnerability may not be one that displays data but will display differently depending on the results of a logical statement injected into the legitimate SQL statement called for that page. This type of attack can become time-intensive because a new statement must be crafted for each bit recovered. There are several tools that can automate these attacks once the location of the vulnerability and the target information has been established

Conditional responses

One type of blind SQL injection forces the database to evaluate a logical statement on an ordinary application screen. As an example, a book review website uses a query string to determine which book review to display. So the URL http://books.example.com/showReview.php?ID=5 would cause the server to run the query

```
SELECT * FROM bookreviews WHERE ID = 'Value(ID)';
```

from which it would populate the review page with data from the review with ID 5, stored in the table bookreviews. The query happens completely on the server; the user does not know the names of the database, table, or fields, nor does the user know the query string. The user only sees that the above URL returns a book review. A hacker can load the URLs http://books.example.com/showReview.php?ID=5 OR 1=1 and http://books.example.com/showReview.php?ID=5 AND 1=2, which may result in queries

```
SELECT * FROM bookreviews WHERE ID = '5' OR '1'='1';
SELECT * FROM bookreviews WHERE ID = '5' AND '1'='2';
```

espectively. If the original review loads with the "1=1" URL and a blank or error page is returned from the "1=2" URL, and the returned page has not been created to alert the user the input is invalid, or in other words, has been caught by an input test script, the site is likely vulnerable to a SQL injection attack as the query will likely have passed through successfully in both cases. The hacker may proceed with this query string designed to reveal the version number of MySQL running on the server: http://books.example.com/showReview.php?ID=5 AND substring(@@version,1,1)=4, which would show the book review on a server running MySQL 4 and a blank or error page otherwise. The hacker can continue to use code within query strings to glean more information from the server until another avenue of attack is discovered or his or her goals are achieved.

Second Order SQL Injection

Second Order SQLi happens when submitted values contain SQL injection attacks that are stored instead of executed immediately. In some cases, the application may correctly encode a SQL statement and store it as valid SQL. Then, another part of that application without controls to protect against SQL Injection might execute that stored SQL statement. This attack requires more knowledge of how submitted values are later used. Automated web application security scanners would not easily detect this type of SQL Injection and may need to be manually instructed where to check for evidence of second order SQLi.

Mitigation Techniques

1) Parameterized statements / Prepared statements

With most development platforms, parameterized statements that work with parameters can be used (sometimes called placeholders or bind variables) instead of embedding user input in the statement. A placeholder can only store a value of the given type and not an arbitrary SQL fragment. Hence the SQL injection would simply be treated as a strange (and probably invalid) parameter value.

In many cases, the SQL statement is fixed, and each parameter is a scalar, not a table. The user input is then assigned (bound) to a parameter.

For more information, read Article No. 77.

2) Enforcement at the coding level

Using object-relational mapping libraries avoids the need to write SQL code. The ORM library in effect will generate parameterized SQL statements from object-oriented code.

3) Escaping

A straightforward, though error-prone, way to prevent injections is to escape characters that have a special meaning in SQL. The manual for an SQL DBMS explains which characters have a special meaning, which allows creating a comprehensive blacklist of characters that need translation. For instance, every occurrence of a single quote (') in a parameter must be replaced by two single quotes ('') to form a valid SQL string literal. For example, in PHP it is usual to escape parameters using the function mysqli_real_escape_string(); before sending the SQL query:

```
$mysqli = new mySqli('hostname', 'db_username', 'db_password', 'db_name');
$query = sprintf("SELECT * FROM `Users` WHERE UserName='%s' AND Password='%s'",
$mysqli->real_escape_string($Username),
$mysqli->real_escape_string($Password));
$mysqli->query($query);
```

Note: 'mysql' is deprecated, and should be avoided. `mysqli` is preferred. This function, i.e. mysql_real_escape_string(), calls MySQL's library function mysql_real_escape_string, which prepends backslashes to the following characters: \x00, \n, \r, \, ', " and \x1a. This function must always (with few exceptions) be used to make data safe before sending a query to MySQL.

There are other functions for many database types in PHP such as pg_escape_string() for PostgreSQL. There is, however, one function that works for escaping characters, and is used especially for querying on databases that do not have escaping functions in PHP. This function is: addslashes(string $str). It returns a string with backslashes before characters that need to be quoted in database queries, etc. These characters are single quote ('), double quote ("), backslash (\) and NUL (the NULL byte).

Routinely passing escaped strings to SQL is error prone because it is easy to forget to escape a given string. Creating a transparent layer to secure the input can reduce this error-proneness, if not entirely eliminate it.

In .Net Framework just use parameterized query or stored procedure using parameters

```
SqlCommand cmd = new SqlCommand("Select * from users where username = @paramUser and password = @paramPass", conn);
cmd.CommandType = CommandType.Text;
cmd.Parameters.Add("@paramUser", SqlDbType.Nvarchar, 20).Value = Username;
cmd.Parameters.Add("@paramPass", SqlDbType.Nvarchar, 20).Value = Password;
```

In this case if for example Username="' or 1=1--" this will not succeed.

4) Pattern check

Integer, float or boolean parameters can be checked if their value is valid representation for the given type. Strings that must follow some strict pattern (date, UUID, alphanumeric only, etc.) can be checked if they match this pattern.

5) Database permissions

Limiting the permissions on the database logon used by the web application to only what is needed may help reduce the effectiveness of any SQL injection attacks that exploit any bugs in the web application.

For example, on Microsoft SQL Server, a database logon could be restricted from selecting on some of the system tables which would limit exploits that try to insert JavaScript into all the text columns in the database.

```
deny select on sys.sysobjects to webdatabaselogon;
deny select on sys.objects to webdatabaselogon;
deny select on sys.tables to webdatabaselogon;
deny select on sys.views to webdatabaselogon;
deny select on sys.packages to webdatabaselogon;
```

Posted by: Bhavesh Patel - Sun, Jul 27, 2014 at 12:00 AM This article has been viewed 464 times.
Filed Under: Web

Article Rating (1 Votes)

★★★★☆

Rate this article
Select One ▾ Rate

Bookmark Article (CTRL-D)

Attachments

There are no attachments for this article.

Related Articles

Web Application - Architecture
Viewed 378 times since Wed, Oct 9, 2013

Different Types of Hacking Techniques
Viewed 500 times since Tue, Aug 27, 2013