Research Article

iMedPub Journals
www.imedpub.com

American Journal of Computer Science and Information Technology

2021
Vol.9 No.8:106

Overview of Sql Injection

Saurav Misra*

Department of Computer Science and Information Technology, India

***Corresponding author:** Misra S, Department of Computer Science and Information Technology, India; Tel No: 7032901764; E-Mail: saurav.misra89@gmail.com

Received date: July 25, 2020; **Accepted date:** August 19, 2021; **Published date:** August 30, 2021

Citation: Misra S (2020) Overview of Sql Injection. Am J Compt Sci Inform Technol Vol.9 No.8:106

Abstract

SQL Injection is a major injection technique, which is used to attack data-driven Applications.

Procedures and functions that use dynamic SQL queries by concatenating the text inputs to the dynamic SQL are prone to SQL Injection attack as someone can provide extra commands/malicious text through the input parameter and when executed can result in the unexpected results.

Example

```
DECLARE @sqlText nvarchar(MAX), @user_id nvarchar(MAX);
SET @sqlText = 'SELECT * FROM Users WHERE user_id = '+
@user_id;
EXEC(@sqlText);
```

Here, if the user will provide @user_id = '105; DROP TABLE SomeTable', an unexpected DROP table will happen.

Best Practices to prevent SQL Injection

Execute Dynamic SQL queries, using SP_EXECUTESQL procedure with the parameters.

While writing dynamic SQL queries and executing them, one needs to be cautious in regards to the following.

1. Avoid concatenating the parameter variables with the query.

Example

```
declare @cmd nvarchar(MAX)
set @cmd = N 'select * from dbo.MyTable where col1 = ' +
@arg1;

--The above query needs to be rewritten as:
declare @cmd nvarchar(MAX);
declare @parameters nvarchar(MAX);
set @cmd = N 'select * from dbo.MyTable where col1 = @arg
1';
set @parameters = '@arg1 VARCHAR(MAX)';
```

2. Avoid executing dynamic SQL queries, using EXEC stored procedure. This approach does not support passing of parameters.

Always use SP_EXECUTESQL procedure with the parameters to execute dynamic SQL queries.

Example

Let @arg1 be the parameter supplied to the procedure, which contains the script, mentioned below.

```
declare @cmd nvarchar(MAX);
set @cmd = N 'select * from dbo.MyTable where col1 = ' +
@arg1;
EXEC(@cmd);
--The above command should be rewritten as:
declare @cmd nvarchar(MAX);
declare @parameters nvarchar(MAX);
set @cmd = N 'select * from dbo.MyTable where col1 = @arg
1';
set @parameters = '@arg1 VARCHAR(MAX)';
```

EXEC SP_EXECUTESQL

```
EXEC SP_EXECUTESQL
@cmd, --dynamic sql command as the first parameter
@parameters, --
definition of parameters as the second parameter
@arg1 = @arg1;
--Assign the parameter value to the parameter
```

Note

- 1. The first 2 parameters of SP_EXECUTESQL (@cmd and @parameters should always be of type nvarchar.
- 2. If the dynamic SQL requires multiple string parameters, the parameters can be written separated by commas.

Example

```
Declare
@cmd nvarchar(MAX),
    @params nvarchar(MAX),
    @arg1VARCHAR(MAX) = 'ParamValue1',
    @arg2VARCHAR(MAX) = ' ParamValue2';
SET @cmd = 'SELECT * FROM dbo.MyTable WHERE col1=@arg1 an
d col2=@arg2';
SET @params = '@arg1 VARCHAR(MAX),@arg2 VARCHAR(MAX)';

--Execute the above query
EXEC SP_EXECUTESQL
@cmd,
@params,
@arg1 = @arg1,
@arg2 = @arg2;
```

Here, is a complete example, which demonstrates the usage of dynamic SQL in a stored procedure in the correct way.

```
create procedure test_procedure1(@arg1 VARCHAR(MAX))
as
  declare @cmd nvarchar(MAX)
  declare @parameters nvarchar(MAX)
  set @cmd = N 'select * from dbo.MyTable where col1 = @a
rg1'
  set @parameters = '@arg1 VARCHAR(MAX)'
  EXEC sp_Executesql @cmd, @parameters, @arg1 = @arg1;
go
  Executing the procedure
      declare @argVARCHAR(MAX);
      SET @arg = 'Some Text';
      EXEC test_procedure1 @arg;
      go
```

Guidelines to follow while using parameter in like clause in dynamic SQL

when we use the parameters supplied to a procedure in a dynamic SQL command and execute it, using EXEC procedure, there is a chance the input parameter can be used to hack into the database object.

Example

```
declare
@cmd nvarchar(MAX),
    @search_string varchar(100);

SET @search_string = '1234';
SET @cmd = 'SELECT * FROM dbo.MyServers WHERE server_name LIKE
 ''%' + @search_string + '%''';

EXEC(@cmd);
```

This works fine but if I pass something like this as @search_string, the code will be as follows.

```
declare
@cmd nvarchar(MAX),
    @search_string varchar(100);

SET @search_string = 'u'' OR 1=1 --';
SET @cmd = 'SELECT * FROM dbo.MyServers WHERE server_name LIKE
 ''%' + @search_string + '%''';

EXEC(@cmd);
```

This will list out every record from the dbo.My Servers table as the command, which will go to the db. Will be.

SELECT * FROM dbo.MyServers WHERE server_name LIKE '%u' OR 1=1 --%'

Here, the best practice is to embed the parameters (search string) in the dynamic SQL command and execute it, using SP_EXECUTESQL with the parameters, as shown below

```
declare
@cmd nvarchar(MAX),
    @params nvarchar(MAX),
    @search_string varchar(100);

SET @search_string = '1234';
--SET @search_string = 'u'' OR 1=1 --';
SET @cmd = 'SELECT * FROM dbo.MyServers WHERE server_name LIKE
 ''%''+' + '@search_string' + '+''%''';
SET @params = '@search_string varchar(100)';
     EXEC sp_executesql
     @cmd,
     @params,
     @search_string = @search_string;
```

If the supplied pattern matches, the query upon execution will generate the appropriate records.

If a malicious pattern is supplied, the execution will result in an empty result set . Please follow the example, stated below.

```
declare
    @cmd nvarchar(MAX),
    @params nvarchar(MAX),
    @search_string varchar(100);

--SET @search_string = '1234';
SET @search_string = 'u'' OR 1=1 --';
SET @cmd = 'SELECT * FROM dbo.MyServers WHERE server_name LIKE
 ''%''+' + '@search_string' + '+''%''';
SET @params = '@search_string varchar(100)';
     EXEC sp_executesql
     @cmd,
     @params,
     @search_string = @search_string;
```

This will result in an empty resultset and our data will not show up.

Guidelines to use table/column names in dynamic SQL:

While using the table/column names as the parameters in a dynamic SQL command, the system defined function QUOTENAME should be used to enclose the table/column name with in [and].

Example

```
SELECT QUOTENAME('MyServers');
Output: [MyServers]

Please check out the below examples:

declare @tabname nvarchar(100), @sql nvarchar(MAX);
SET @tabname = 'MyServers;DROP TABLE dbo.MyConfigs';
SET @sql = 'SELECT top 10 * FROM ' + @tabname;
EXEC(@sql);
     GO
```

Here, the @tabname variable can be used to manipulate the database in a wrong way. To prevent it, @tabname should be enclosed within [and] as in this case [My Servers; drop table dbo.My Configs] will not be considered as a valid table name.

Here is the script

```
declare @tabname nvarchar(100), @sql nvarchar(MAX);
SET @tabname = 'MyServers;PRINT ''HELLO''';
SET @sql = 'SELECT top 10 * FROM ' + QUOTENAME(@tabname);

EXECSP_EXECUTESQL @sql;
```

Output

Invalid Object name 'MyServers;PRINT 'HELLO''.

Here is another example, where both column and table names are used in a dynamic SQL query.

Someone can push something dangerous through the column name.

```
declare @tabname nvarchar(100), @column nvarchar(100), @sql nv
archar(MAX);
SET @tabname = 'MyServers';
SET @column = 'server_name FROM dbo.MyServers;PRINT ''HELLO BR
O! U R HACKED''--';
SET @sql = 'SELECT ' + @column + ' FROM ' + QUOTENAME(@tabname
);
EXEC(@sql);
GO
```

This will print out all the Server names from your dbo.MyServers table.

This should be rewritten, as stated below.

```
declare
    @tabname nvarchar(100),
    @column nvarchar(100),
    @sql nvarchar(MAX);
SET @tabname = 'MyServers';
SET @column = 'server_name FROM dbo.MyServers;PRINT ''HELLO BR
O! U R HACKED''--';
SET @sql = 'SELECT ' + QUOTENAME(@column) + ' FROM ' + QUOTENA
ME(@tabname);
EXECSP_EXECUTESQL @sql;
GO
```

Output

Invalid column name 'server_name FROM dbo.MyServers;PRINT 'HELLO BRO! U R HACKED

REFERENCES

1. Aarafat Aldhaqm, Shukor Razak, Siti Othman, Abdulalem Aldolah, and Md Ngadi (2016). Conceptual Investigation Process Model for Managing Database Forensic Investigation Knowledge. 12 (02 2016), 386–394.
2. Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan (2007). CANDID: Preventing Sql Injection Attacks Using Dynamic Candidate Evaluations. In Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07). ACM, New York, NY, USA, 12–24.
3. Martin Bravenboer, Eelco Dolstra, and Eelco Visser (2007). Preventing Injection Attacks with Syntax Embeddings. In Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE '07). ACM, New York, NY, USA, 3–12.
4. Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti (2005). Using Parse Tree Validation to Prevent SQL Injection Attacks. In Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM '05). ACM, New York NY, USA, 106–113.
5. L. Chen and L. Tao (2011). Teaching Web Security Using Portable Virtual Labs. In 2011 IEEE 11th International Conference on Advanced Learning Technologies. 491–495.