**SQL Injection**

SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. Any procedure that constructs SQL statements should be reviewed for injection vulnerabilities because SQL Server will execute all syntactically valid queries that it receives. Even parameterized data can be manipulated by a skilled and determined attacker.

The primary form of SQL injection consists of direct insertion of code into user-input variables that are concatenated with SQL commands and executed. A less direct attack injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code is executed.

The injection process works by prematurely terminating a text string and appending a new command. Because the inserted command may have additional strings appended to it before it is executed, the malefactor terminates the injected string with a comment mark "--". Subsequent text is ignored at execution time.

The following script shows a simple SQL injection. The script builds an SQL query by concatenating hard-coded strings together with a string entered by the user:

var Shipcity;

ShipCity = Request.form ("ShipCity");

var sql = "select \* from OrdersTable where ShipCity = '" + ShipCity + "'";

The user is prompted to enter the name of a city. If she enters Redmond, the query assembled by the script looks similar to the following:

SELECT \* FROM OrdersTable WHERE ShipCity = 'Redmond'

However, assume that the user enters the following:

Redmond'; drop table OrdersTable--

In this case, the following query is assembled by the script:

SELECT \* FROM OrdersTable WHERE ShipCity = 'Redmond';drop table OrdersTable--'

The semicolon (;) denotes the end of one query and the start of another. The double hyphen (--) indicates that the rest of the current line is a comment and should be ignored. If the modified code is syntactically correct, it will be executed by the server. When SQL Server processes this statement, SQL Server will first select all records in OrdersTable where ShipCity is Redmond. Then, SQL Server will drop OrdersTable.

As long as injected SQL code is syntactically correct, tampering cannot be detected programmatically. Therefore, you must validate all user input and carefully review code that executes constructed SQL commands in the server that you are using. Coding best practices are described in the following sections in this topic.

http://i.msdn.microsoft.com/Global/Images/clear.gif Validate All Input

Always validate user input by testing type, length, format, and range. When you are implementing precautions against malicious input, consider the architecture and deployment scenarios of your application. Remember that programs designed to run in a secure environment can be copied to an nonsecure environment. The following suggestions should be considered best practices:

* Make no assumptions about the size, type, or content of the data that is received by your application. For example, you should make the following evaluation:
  + How will your application behave if an errant or malicious user enters a 10-megabyte MPEG file where your application expects a postal code?
  + How will your application behave if a DROP TABLE statement is embedded in a text field?
* Test the size and data type of input and enforce appropriate limits. This can help prevent deliberate buffer overruns.
* Test the content of string variables and accept only expected values. Reject entries that contain binary data, escape sequences, and comment characters. This can help prevent script injection and can protect against some buffer overrun exploits.
* When you are working with XML documents, validate all data against its schema as it is entered.
* Never build Transact-SQL statements directly from user input.
* Use stored procedures to validate user input.
* In multitiered environments, all data should be validated before admission to the trusted zone. Data that does not pass the validation process should be rejected and an error should be returned to the previous tier.
* Implement multiple layers of validation. Precautions you take against casually malicious users may be ineffective against determined attackers. A better practice is to validate input in the user interface and at all subsequent points where it crosses a trust boundary.   
  For example, data validation in a client-side application can prevent simple script injection. However, if the next tier assumes that its input has already been validated, any malicious user who can bypass a client can have unrestricted access to a system.
* Never concatenate user input that is not validated. String concatenation is the primary point of entry for script injection.
* Do not accept the following strings in fields from which file names can be constructed: AUX, CLOCK$, COM1 through COM8, CON, CONFIG$, LPT1 through LPT8, NUL, and PRN.

When you can, reject input that contains the following characters.

|  |  |
| --- | --- |
| **Input character** | **Meaning in Transact-SQL** |
| ; | Query delimiter. |
| ' | Character data string delimiter. |
| -- | Comment delimiter. |
| /\* ... \*/ | Comment delimiters. Text between /\* and \*/ is not evaluated by the server. |
| **xp\_** | Used at the start of the name of catalog-extended stored procedures, such as **xp\_cmdshell**. |

**Use Type-Safe SQL Parameters**

The **Parameters** collection in SQL Server provides type checking and length validation. If you use the **Parameters** collection, input is treated as a literal value instead of as executable code. An additional benefit of using the **Parameters** collection is that you can enforce type and length checks. Values outside the range will trigger an exception. The following code fragment shows using the **Parameters** collection:

SqlDataAdapter myCommand = new SqlDataAdapter("AuthorLogin", conn);

myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;

SqlParameter parm = myCommand.SelectCommand.Parameters.Add("@au\_id",

SqlDbType.VarChar, 11);

parm.Value = Login.Text;

In this example, the @au\_id parameter is treated as a literal value instead of as executable code. This value is checked for type and length. If the value of @au\_id does not comply with the specified type and length constraints, an exception will be thrown.

**Use Parameterized Input with Stored Procedures**

Stored procedures may be susceptible to SQL injection if they use unfiltered input. For example, the following code is vulnerable:

SqlDataAdapter myCommand =

new SqlDataAdapter("LoginStoredProcedure '" +

Login.Text + "'", conn);

If you use stored procedures, you should use parameters as their input.

**Use the Parameters Collection with Dynamic SQL**

If you cannot use stored procedures, you can still use parameters, as shown in the following code example:

SqlDataAdapter myCommand = new SqlDataAdapter(

"SELECT au\_lname, au\_fname FROM Authors WHERE au\_id = @au\_id", conn);

SQLParameter parm = myCommand.SelectCommand.Parameters.Add("@au\_id",

SqlDbType.VarChar, 11);

Parm.Value = Login.Text;

**Filtering Input**

Filtering input may also be helpful in protecting against SQL injection by removing escape characters. However, because of the large number of characters that may pose problems, this is not a reliable defense. The following example searches for the character string delimiter.

private string SafeSqlLiteral(string inputSQL)

{

return inputSQL.Replace("'", "''");

}

**LIKE Clauses**

Note that if you are using a LIKE clause, wildcard characters still must be escaped:

s = s.Replace("[", "[[]");

s = s.Replace("%", "[%]");

s = s.Replace("\_", "[\_]");

http://i.msdn.microsoft.com/Global/Images/clear.gif Reviewing Code for SQL Injection

You should review all code that calls EXECUTE, EXEC, or **sp\_executesql**. You can use queries similar to the following to help you identify procedures that contain these statements. This query checks for 1, 2, 3, or 4 spaces after the words EXECUTE or EXEC.

SELECT object\_Name(id) FROM syscomments

WHERE UPPER(text) LIKE '%EXECUTE  (%'

OR UPPER(text) LIKE '%EXECUTE  (%'

OR UPPER(text) LIKE '%EXECUTE   (%'

OR UPPER(text) LIKE '%EXECUTE    (%'

OR UPPER(text) LIKE '%EXEC (%'

OR UPPER(text) LIKE '%EXEC  (%'

OR UPPER(text) LIKE '%EXEC   (%'

OR UPPER(text) LIKE '%EXEC    (%'

OR UPPER(text) LIKE '%SP\_EXECUTESQL%'

**Wrapping Parameters with QUOTENAME() and REPLACE()**

In each selected stored procedure, verify that all variables that are used in dynamic Transact-SQL are handled correctly. Data that comes from the input parameters of the stored procedure or that is read from a table should be wrapped in QUOTENAME() or REPLACE(). Remember that the value of **@variable** that is passed to QUOTENAME() is of **sysname**, and has a maximum length of 128 characters.

|  |  |
| --- | --- |
| **@variable** | **Recommended wrapper** |
| Name of a securable | QUOTENAME(**@variable**) |
| String of ≤ 128 characters | QUOTENAME(**@variable,** '''') |
| String of > 128 characters | REPLACE(**@variable**,'''', '''''') |

When you use this technique, a SET statement can be revised as follows:

--Before:

SET @temp = N'select \* from authors where au\_lname='''

+ @au\_lname + N''''

--After:

SET @temp = N'select \* from authors where au\_lname='''

+ REPLACE(@au\_lname,'''','''''') + N''''

**Injection Enabled by Data Truncation**

Any dynamic Transact-SQL that is assigned to a variable will be truncated if it is larger than the buffer allocated for that variable. An attacker who is able to force statement truncation by passing unexpectedly long strings to a stored procedure can manipulate the result. For example, the stored procedure that is created by the following script is vulnerable to injection enabled by truncation.

CREATE PROCEDURE sp\_MySetPassword

@loginname sysname,

@old sysname,

@new sysname

AS

-- Declare variable.

-- Note that the buffer here is only 200 characters long.

DECLARE @command varchar(200)

-- Construct the dynamic Transact-SQL.

-- In the following statement, we need a total of 154 characters

-- to set the password of 'sa'.

-- 26 for UPDATE statement, 16 for WHERE clause, 4 for 'sa', and 2 for

-- quotation marks surrounded by QUOTENAME(@loginname):

-- 200 – 26 – 16 – 4 – 2 = 154.

-- But because @new is declared as a sysname, this variable can only hold

-- 128 characters.

-- We can overcome this by passing some single quotation marks in @new.

SET @command= 'update Users set password=' + QUOTENAME(@new, '''') + ' where username=' + QUOTENAME(@loginname, '''') + ' AND password = ' + QUOTENAME(@old, '''')

-- Execute the command.

EXEC (@command)

GO

By passing 154 characters into a 128 character buffer, an attacker can set a new password for sa without knowing the old password.

EXEC sp\_MySetPassword 'sa', 'dummy', '123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012'''''''''''''''''''''''''''''''''''''''''''''''''''

For this reason, you should use a large buffer for a command variable or directly execute the dynamic Transact-SQL inside the EXECUTE statement.

**Truncation When QUOTENAME(@variable, '''') and REPLACE() Are Used**

Strings that are returned by QUOTENAME() and REPLACE() will be silently truncated if they exceed the space that is allocated. The stored procedure that is created in the following example shows what can happen.

CREATE PROCEDURE sp\_MySetPassword

@loginname sysname,

@old sysname,

@new sysname

AS

-- Declare variables.

DECLARE @login sysname

DECLARE @newpassword sysname

DECLARE @oldpassword sysname

DECLARE @command varchar(2000)

-- In the following statements, the data stored in temp variables

-- will be truncated because the buffer size of @login, @oldpassword,

-- and @newpassword is only 128 characters, but QUOTENAME() can return

-- up to 258 characters.

SET @login = QUOTENAME(@loginname, '''')

SET @oldpassword = QUOTENAME(@old, '''')

SET @newpassword = QUOTENAME(@new, '''')

-- Construct the dynamic Transact-SQL.

-- If @new contains 128 characters, then @newpassword will be '123... n

-- where n is the 127th character.

-- Because the string returned by QUOTENAME() will be truncated,

-- it can be made to look like the following statement:

-- UPDATE Users SET password ='1234. . .[127] WHERE username=' -- other stuff here

SET @command = 'UPDATE Users set password = ' + @newpassword

+ ' where username =' + @login + ' AND password = ' + @oldpassword;

-- Execute the command.

EXEC (@command)

GO

Therefore, the following statement will set the passwords of all users to the value that was passed in the previous code.

EXEC sp\_MyProc '--', 'dummy', '12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678'

You can force string truncation by exceeding the allocated buffer space when you use REPLACE(). The stored procedure that is created in the following example shows what can happen.

CREATE PROCEDURE sp\_MySetPassword

@loginname sysname,

@old sysname,

@new sysname

AS

-- Declare variables.

DECLARE @login sysname

DECLARE @newpassword sysname

DECLARE @oldpassword sysname

DECLARE @command varchar(2000)

-- In the following statements, data will be truncated because

-- the buffers allocated for @login, @oldpassword and @newpassword

-- can hold only 128 characters, but QUOTENAME() can return

-- up to 258 characters.

SET @login = REPLACE(@loginname, '''', '''''')

SET @oldpassword = REPLACE(@old, '''', '''''')

SET @newpassword = REPLACE(@new, '''', '''''')

-- Construct the dynamic Transact-SQL.

-- If @new contains 128 characters, @newpassword will be '123...n

-- where n is the 127th character.

-- Because the string returned by QUOTENAME() will be truncated, it

-- can be made to look like the following statement:

-- UPDATE Users SET password='1234…[127] WHERE username=' -- other stuff here

SET @command= 'update Users set password = ''' + @newpassword + ''' where username='''

+ @login + ''' AND password = ''' + @oldpassword + '''';

-- Execute the command.

EXEC (@command)

GO

As with QUOTENAME(), string truncation by REPLACE() can be avoided by declaring temporary variables that are large enough for all cases. When possible, you should call QUOTENAME() or REPLACE() directly inside the dynamic Transact-SQL. Otherwise, you can calculate the required buffer size as follows. For @outbuffer = QUOTENAME(@input), the size of @outbuffer should be 2\*(len(@input)+1). When you use REPLACE() and doubling quotation marks, as in the previous example, a buffer of 2\*len(@input) is enough.

The following calculation covers all cases:

While len(@find\_string) > 0, required buffer size =

round(len(@input)/len(@find\_string),0) \* len(@new\_string)

+ (len(@input) % len(@find\_string))

**Truncation When QUOTENAME(@variable, ']') Is Used**

Truncation can occur when the name of a SQL Server securable is passed to statements that use the form QUOTENAME(@variable, ']'). The following example shows this.

CREATE PROCEDURE sp\_MyProc

@schemaname sysname,

@tablename sysname,

AS

-- Declare a variable as sysname. The variable will be 128 characters.

-- But @objectname actually must allow for 2\*258+1 characters.

DECLARE @objectname sysname

SET @objectname = QUOTENAME(@schemaname)+'.'+ QUOTENAME(@tablename)

-- Do some operations.

GO

When you are concatenating values of type **sysname**, you should use temporary variables large enough to hold the maximum 128 characters per value. If possible, call QUOTENAME() directly inside the dynamic Transact-SQL. Otherwise, you can calculate the required buffer size as explained in the previous section.