SQL injection is a well known web application vulnerability. The attacker's aim is to execute his own SQL code on the victim's database through a web application. The attack is similar to XSS. The attacker sends a string to the web application via a form or URL parameter. If that string is used as a dynamic part of an SQL query (for example as part of a WHERE condition) and not protected properly, the attacker can modify the query before it is executed.

We can divide vulnerabilities into two kinds – *classic* and *blind*:

- Classic SQL injection - the attacker can see the real error message from the SQL server.
- Blind SQL injection - the attacker sees only a general error page. These injections are harder to exploit because the attackers do not know how exactly they can inject the code. The attackers usually use enumeration in this type of attack. Then, according to the time based errors or the displayed error message can they determine which of their queries have passed and which did not.

Example of SQL injection

A simple web page with a textbox and button which is used for searching users:

```
<asp:TextBox ID="txtUserName" runat="server"></asp:TextBox>
<asp:Button ID="btnSearch" runat="server" Text="Search" onclick="btnSearch_Click" />
```

In the code behind, the page has the following code:

```
using System.Data;

using CMS.Membership;
using CMS.Helpers;

protected void btnSearch_Click(object sender, EventArgs e)
{
        DataSet ds = UserInfoProvider.GetUsers().Where("UserName LIKE '%" +
txtUserName.Text + "%'");
        if (!DataHelper.DataSourceIsEmpty(ds))
        {
                Response.Write(ds.Tables[0].Rows[0]["FullName"]);
        }
}
```

Now if a user inserts something like "admin", the user gets the full name of the global administrator on the output. The SQL query looks like this:

```
SELECT * FROM CMS_User WHERE UserName LIKE '%admin%'
```

This is a correct query which doesn't cause any problems. But if the user inserts something like "**a'; DROP table CMS_User --**", the resulting query is:

```
SELECT * FROM CMS_User WHERE UserName LIKE '%a'; DROP table CMS_User --%'
```

The query is executed, resulting in a deleted table.

What can SQL injection attacks do

With an SQL injection vulnerability, the attacker can do exactly the same operations with the database as the web application itself. For example, the attacker can read and modify all data or the database schema. Also, T-SQL supports the xp_cmdshell() function, which executes operating system commands. So, the attacker can basically manage the whole server.

Finding SQL injection vulnerabilities

The first technique is based on trying. Insert strings like:

- 'whatever – basic test,
- DROP,
- something,

to all inputs and URL parameters. Do not test only the apostrophe character (in the next chapter, you will see that you can exploit an application even without the apostrophe character).

The second way is to search for vulnerabilities in code. You can search for methods executing SQL queries. Then, you can check variables which are inputs of these methods. The aim of checking is searching for protection against SQL injection.

Avoiding SQL injection in Kentico

There are many ways of protection against SQL injection. Kentico provides:

- DataQuery
- Query objects
- SQL Parameters
- Apostrophe escaping

DataQuery

The best way to avoid SQL injection is to use the default Kentico API provider methods when loading data (ObjectQuery / DataQuery). The key security aspect of DataQuery is in processing parameter-driven queries.

To create WHERE conditions, we strongly recommend using specific *Where* methods for the given purpose, such as:

- WhereContains
- WhereNull
- WhereEquals
- WhereStartsWith

To get the full list of the Where methods, see the IWhereCondition reference.

Example

```
using CMS.Membership;

...

String avatarName = txtAvatarname.Text;

// Loads avatar objects whose "AvatarName" column starts with a specific name
ObjectQuery<AvatarInfo> avatars = AvatarInfoProvider.GetAvatars().WhereStartsWith
("AvatarName", avatarName);
```

Query objects

If you need to execute queries manually, use the appropriate **query object classes** from the Kentico *CMS.DataEngine* namespace to build parts of the query:

- **WhereCondition**
- **Columns**
- **OrderBy**

Example

```
using CMS.DataEngine;

...

// Builds a WHERE condition for loading documents
string where = new WhereCondition()
    .WhereEquals("AttachmentDocumentID", node.DocumentID)
    .WhereStartsWith("AttachmentName", searchFileName).ToString(true);
```

SQL Parameters

For WHERE conditions in INSERT, UPDATE and DELETE query types and stored procedures, handle parameters via **QueryDataParameters** objects.

Example

```
using CMS.DataEngine;

...

QueryDataParameters parameters = new QueryDataParameters();
parameters.Add("@param", param);
```

The SQL server replaces **@param** with your value. The value is treated as a literal, which means that even if your value contains a piece of SQL code, the SQL server **does not execute the code**.

Queries with the exec() function

Parameters are almost 100% secure. But if you build a query, which is executed with the built-in **exec()** function, the parameters are processed the standard way (the query is executed with them, even if they contain malicious SQL code).

The following is an example of an **unsafe query**:

Wrong

```
CREATE PROCEDURE injection( @param varchar(30) )
AS
SET NOCOUNT ON
DECLARE @query VARCHAR(100)
SET @query = 'SELECT * FROM ' + @param
exec(@query)
GO
```

Apostrophe escaping

The last protection technique is manually replacing the dangerous apostrophe character with an escape sequence of two apostrophes. In code, you often build WHERE conditions for SELECT queries. When a part of the condition is a dynamically obtained string (e.g., from the database, input by a user, etc.), you must enclose it with apostrophes and perform replacing.

You can use two different options:

- To escape values in ***equals*** query patterns, use the **SqlHelper.EscapeQuotes** method:

```
string text = txtAvatarname.Text;

string where = "AvatarName = N'" + SqlHelper.EscapeQuotes(text) + "'";
```

- To escape values in ***LIKE*** query patterns, use both the **SqlHelper.EscapeQuotes** and **SqlHelper.EscapeLikeValue** methods:

```
string text = txtAvatarname.Text;

string where = "AvatarName LIKE N'%" + SqlHelper.EscapeLikeText(SqlHelper.
EscapeQuotes(text)) + "%'";
```

The *Sqlhelper.EscapeQuotes* method ensures that all apostrophes are escaped to avoid SQL injection attacks. The second method *SqlHelper.EscapeLikeText* escapes wildcards used in SQL LIKE syntax — square brackets, percentage and underscore symbols.

This is a correct solution, but **only for string values**. Never use apostrophe escaping for other data types than strings. For example:

Wrong

```
Guid guid = ... ;
string where = "SomeGUID = '" + guid.ToString().Replace("'", "''") + "'";
```

In this example, replacing is unnecessary because GUID values can only contain letters and numbers in a specific format.

A worse situation can occur when you believe that you are using non-string data types (for example int), but the variable you are actually using is a string:

Wrong

```
string id = ... ;
string where = "SomeID = " + id.Replace("'", "''");
```

There are two reasons why this code is wrong:

- It is possible to execute malicious code even without the apostrophe character. In the previous examples, you needed apostrophes to inject string constants into the SQL command, but there are no enclosing apostrophes for integer values.
- In SQL, the Char() function converts numeric values to their ASCII representations. And these ASCII letters can be concatenated, so the attacker can write anything into the query.

To solve such situations, convert the values to the correct data type, for example:

Correct

```
using CMS.Helpers;

...

string id = ... ;
string where = "SomeID = " + ValidationHelper.GetInteger(id, 0);
```

Summary

- Use the Kentico provider methods to load data whenever possible (ObjectQuery / DataQuery)
- Protect dynamic parts in INSERT, UPDATE and DELETE queries with SQL parameters.
- Don't ever use the **exec()** function in your SQL code in combination with parameters.
- When you build a SELECT query in code, all used strings taken from external sources must be protected with the **SqlHelper.EscapeQuotes** and **SqlHelper.EscapeLikeValue** methods or use SQL parameters.
- Always escape values from arrays (or lists etc.) when you save them into strings.
- Never rely on JavaScript validation. JavaScript is executed on the client side so the attacker can disable validation.
- When you work with other data types than strings, always convert the data to the given type or validate the value via regular expressions.