Macros and security

Macros are an essential part of Kentico CMS. You can read more about them in the Macro expressions chapter. This topic focuses on the security mechanisms and best practices related to macros.

On this page

Related pages

- Macro expressions

Signing macros

Whenever a user saves a complex macro expression, the system automatically adds a security signature. This signature contains the **user name** of the macro's author and a **hash** of the given expression.

You can recognize signed macro expressions by the **#** character, which the system automatically inserts before the closing **%}** parentheses when saving the text containing the macro:

- *{%CurrentSite.SiteID #%}*

A signed macro may look like this in the database:

- *{%CurrentSite.SiteID|(user)administrator|(hash) 9056b7b76629a47a660c40cc2e5b0d92a13d0f9bce178847ca412c3a585552a1%}*

Hashing is omitted when submitting very simple macros (e.g., {%FullName%}). Macros are signed only when they contain an indexer or a dot.

The system checks the macro signature when accessing an object through a different object (for example the user settings through a user, which are two different objects) and evaluates, if the user (identified by the hash) has **permissions** to execute the macro.

Best practice is to submit macros signed by a user who has access to the given objects, but does not have access to anything else (least privilege). The problem is that there are **nested macros** – the result of one macro expression is a different macro expression. This way, a potential attacker could gain access to sensitive information using a macro, which is signed by an administrator and is therefore executed in the context of the administrator.

A simple example of a **nested macro**:

- You have this macro expression on a page: *{% QueryString.GetValue("test", "")#%}*,
- which calls a page with such query parameter: *http://localhost/Final70/Home.aspx?test={% CurrentUser.UserName%}*.

SQL injections

Macros can often become part of SQL queries, for example in *WhereCondition* and *OrderBy* fields of web part properties, which can lead to SQL injection vulnerability.

SQL injection protection in web part properties

Some web part properties are secured against SQL injection attacks, which may affect how macros are resolved in specific cases. By default, this is applied to macros entered into the **WhereCondition** and **OrderBy** web part properties.

If the macro returns a string value that contains single quote characters ('), they will be escaped and replaced by two single quotes (''). This may cause an SQL syntax error if you are using the macro to dynamically insert a part of a query, such as a WHERE clause.

It is possible to disable single quote escaping for a specific macro expression by adding the **handlesqlinjection** parameter and setting its value to *false*:

```
{% ... |(handlesqlinjection)false %}
```

To disable SQL escaping globally for all properties of a specific type of web part, edit its code behind file (e.g. *~/CMSWebParts/Viewers /Documents/cmsrepeater.ascx.cs* for the **Repeater** web part) and add the following line of code into the **SetupControl()** method:

```
this.SQLProperties = "";
```

The **SQLProperties** property is inherited from the **CMSAbstractWebPart** base class by all web parts, but you can override its value to set which properties the system protects.

If you wish to enable SQL escaping for additional web part properties, enter their names into the SQLProperties value separated by semicolons, for example:

```
this.SQLProperties = "wherecondition;orderby;sqlquery";
```

Disabling SQL protection may create security vulnerabilities if the macro resolves its value according to data that can be modified by the website's users, such as in the case of QueryString macros.

SQL injection protection outside of web part properties

If you are using macros outside of web part properties or you have disabled SQL escaping, then you need to use the SQLEscape method to handle SQL macros.

```
WHERE UserName LIKE '%{% SQLEscape(QueryString.GetValue("test", ""))#%}%'
```

However, if you expect a different data type than string (for example an integer), you need to convert the macro to the correct data type. The SQLEscape method is useless in this case.

```
WHERE UserID = {% ToInt(QueryString.GetValue("test", ""), 0)#%}
```

Output encoding of macros

Macro encoding is an essential protection against XSS. You must encode output macros properly, unless you want to display some HTML code using the particular macro. There are several methods for encoding macros in Kentico:

- {% UrlEncode(CurrentUser.FullName)%} - macro is encoded into the query URL.
- {% HTMLEncode(CurrentUser.FullName)%} - macro is a part of standard page output text and text between HTML tags.
- {% JSEncode(CurrentUser.FullName)%} - macro is a part of JS code.
- {% StripTags(CurrentUser.FullName)%} - encodes macros, which are part of standard page output text and text between HTML tags, leaving the the tags out.

Writing custom macro methods

When you create a custom macro method, you have to ensure its security yourself. The macro engine cannot predict, which data are accessed in the method, and thus cannot secure the method.

Unsafe macros

Query macros, cookie macros and data macros (information from the database, e.g. user display name) and their equivalent properties can be potentially dangerous. When you use these macros, you have to secure them against SQL injection and XSS.

It is NOT possible to gain access to user passwords using macros.