Cross site scripting happens when somebody (an attacker) inserts a malicious input into a form (for example, a piece of HTML code). Depending on what happens after that, we divide XSS attacks into these types:

- **Persistent XSS** - a web application (like an instance of Kentico) stores the malicious input in the database. Then, on some other page, the input is rendered. A browser renders the attacker's malicious input as a part of the page's HTML and this is the spot, where the input can cause problems.
- **Non-persistent XSS** - the main difference is that a web application doesn't store the malicious input in the database. Instead, the application renders the input directly as a part of the page's response.
    - A special case of non-persistent XSS is called **DOM-based XSS** - this type of attack is done without sending any requests to the web server. The attacker injects JavaScript code directly.

Types of XSS attacks

Persistent XSS

In this example, we have a standard .aspx page with a textbox, a label and a button:

```
<asp:TextBox runat="server" ID="txtUserName"></asp:TextBox>
<asp:Button runat="server" ID="btnGetMeNameOfUser" Text="Get me name of user" onclick="
btnGetMeNameOfUser_Click" />
<asp:Label runat="server" ID="lblUser"></asp:Label>
```

In code behind, we handle the **OnClick** event of the button. In the handler, we get a UserInfo object from the database. Finally, we render the first name entered into the text box via the label.

```
using CMS.Membership;

...

protected void btnGetMeNameOfUser_Click(object sender, EventArgs e)
{
        UserInfo ui = UserInfoProvider.GetUserInfo(txtUserName.Text);
        if(ui != null)
        {
                lblUser.Text = ui.FirstName;
        }
}
```

Now there is a possibility of XSS attack. The attacker can simply create a user with a first name *"<script>alert(1)</alert>"*. Anyone who sends a request to a page where the attacker's first name would normally be displayed, gets the HTML code injected and executed on their machine (the user gets an alert showing **"1"**).

Non-persistent XSS

Now we have another standard .aspx page with a label:

```
<asp:Label ID="lblInfo" runat="server" EnableViewState="false" CssClass="InfoLabel" />
```

In the code behind, we have the following code:

```
using CMS.Helpers;

...

protected void Page_Load(object sender, EventArgs e)
{
        // Get localization string from URL
        string restring = QueryHelper.GetString("info", "");
        lblInfo.Text = ResHelper.GetString(restring);
}
```

This is an example of a potential vulnerability to a non-persistent XSS attack. The attacker "creates" a URL like *www.ourweb.tld /ourpage.aspx?info=<script>alert(1)</script>*. This URL is sent to a victim. The victim clicks on it, and JavaScript code **alert(1)** gets executed.

DOM-based XSS

Let's have another .aspx page with this JavaScript code:

```
<select>
<script>
        document.write("<OPTION value=1>" + document.location.href.substring(
        document.location.href.indexOf("default=") + 8) + </OPTION>");
        document.write("<OPTION value=2>English</OPTION>");
</script>
</select>
```

When a user navigates to *http://lwww.mydomain.tld/XSSDOMBased.aspx?default=<script>alert(document.cookie)</script>*, the JavaScript code is executed because it is written to the page by the **document.write()** function. The key difference here is that no part of code is handled by the server. The whole attack is done only in the victim's browser.

What can XSS attacks do

In cross site scripting, the attacker can insert any kind of code into a page which is interpreted by the client browser. So, the attacker can, for example:

- change the look of your site,
- change the content of the site,
- insert any JavaScript code to your site,
- redirect the page to an evil one,
- force the users to download malicious code (a virus).

With JavaScript, the attacker can read and steal the user's cookies. With stolen cookies, the attacker can log on to a site (even with an administrator account) without a user name and password. This is the most dangerous thing the attacker can do when your site is XSS vulnerable.

You may think that any at least a little advanced user wouldn't click a link like:

```
blahblah.tld?input=<script>alert('I am going to steal your money');</script>
```

Yeah, but what about a link like:

```
blahblah.tld?input=%3d%3c%73%63%72%69%70%74%3e%6a%61%76%61%73%63%72%69%70%74%28%91%49%
20%61%6d%20%67%6f%69%6e%64%20%74%6f%20%73%74%65%61%6c%20%79%6f%75%72%20%6d%6f%6e%65%79%
92%29%3b%3c%2f%73%63%72%69%70%74%3
```

Can you read that string? It is the same string as the first one, only in hexadecimal encoding, so the browser receives the same text.

Finding XSS vulnerabilities

The first way to find XSS vulnerabilities is based on trying. Simply insert a test string into all inputs and URL parameters. Use strings like:

- <script>alert(1)</script>
- alert(1)
- 'alert(1)

See if your input was executed, if it changed the page or if it caused a JavaScript error. All these signs point to a page vulnerable to XSS.
You can also try to change the HTTP request (for example, if you are using the Firefox browser, you can use add-ons to change the user agent, referrer, etc.).

Another way is to find vulnerabilities in code. You can search for all spots where properties (of Kentico objects) are used and check whether they are HTML encoded when they get rendered on the page. This is the best technique of searching for persistent XSS.

You can also search for spots where context variables (**HTTPContext.XXX**, **HTTPContext**, **Server**, **Request**, **request.querystring** - this one is the most important) or URL parameters (**QueryHelper.GetString()**) are used. This way, you can find non-persistent XSS.
If you want to search for DOM-based XSS, search your code for the following JavaScript objects (these can be influenced by the attacker):

- document.URL
- document.URLUnencoded
- document.location (and many of its properties)
- document.referrer
- window.location

The last way is to use automatic tools for vulnerability searching. These tools are based on similar techniques as manual searching, while they work automatically. However, they often find far too many false positive vulnerabilities and using them is much less effective. The reason for that is simple – these tools (at least those that aren't based on formal verification – about 99% of them) use brute force, while you can use your brain.

Avoiding XSS in Kentico

Why would the attacker be able to perform all types of XSS attacks in the previously specified code examples?

Mainly because the dynamic parts of code (inputs, database data, …) were not encoded properly. So, everything that goes to output (is rendered to HTML) must be properly HTML/JavaScript encoded. What does it mean to properly encode something? Kentico provides the following methods for avoiding XSS attacks:

- **HTMLHelper.HTMLEncode()** - encodes HTML tags, replaces the < and > chars with their HTML entities.
- **QueryHelper.GetText()** - gets a HTML encoded string from the query string.
- **ScriptHelper.GetString()** - replaces special chars like apostrophes (alt+39).
- Correctly keep the type, using for example **QueryHelper.GetInteger()**.
- Use the **HTMLEncode()** transformation function.

Where should you protect your code?

Always on the output before rendering. The reason is that you should secure your web, but you do not want to change the users' input data. You also should not exclusively rely on input validation.

It is also a good practice not to let anyone read cookies, as cookies are usually the main target of XSS attackers. You should make it impossible to access cookies on clients by configuring cookies to be http only – see Web.config file settings.

> ⚠ Server-side encoding cannot protect you from the DOM-based XSS type of attack. Kentico currently does not have any special client-side mechanism (e.g., a JavaScript function) for avoiding DOM-based XSS. There is only one strong recommendation: When you are working with functions/objects that can change output (e.g., *document.write()*, *document.location*, …) in client-side code, do not use the input directly. Let the server encode the potentially malicious input first.

Do not use the WYSIWYG editor on the live site

The editor used in Kentico allows users to input HTML content, including JavaScript.

- If you use the editor for input forms on the live site, you need to properly encode the value before you render it on the website.
- Disable the **Enable WYSIWYG editor** option for any Forums that you have on the live site.

Examples of server-side protection from XSS vulnerabilities

If you have a string value which is taken from the database, you must encode it with **HTMLHelper.HTMLEncode()** before you set it to a control's property (e.g., label text). For example:

```
lblUser.Text = CMS.Helpers.HTMLHelper.HTMLEncode(ui.FirstName);
```

If you have a string value taken from query string (even if it's called id) and you plan to render it to output (i.e. some information message), use **QueryHelper.GetText()** instead of **QueryHelper.GetString()**. For example:

```
string restring = CMS.Helpers.QueryHelper.GetText("info", "");
```

If you are putting some JavaScript into your code, for example:

```
ltScript.Text = CMS.Base.Web.UI.ScriptHelper.GetScript("alert('" + dynamic + "');");
```

Always encode your dynamic parts with **ScriptHelper.GetString(),** even if you have already HTML encoded them. In this case, the attacker does not have to insert any HTML tags (e.g., <script>) because you have already inserted them. The protected code should look like this:

```
ltScript.Text = CMS.Base.Web.UI.ScriptHelper.GetScript(
        "alert('" + CMS.Base.Web.UI.ScriptHelper.GetString(dynamic) + "');") ;
```

Protect your code against XSS when writing **transformations**. You can use the HTMLEncode method and validation methods. For example, you can encode the manufacturer name in product listings:

```
<%# IfEmpty(Eval("SKUManufacturerID"), "-", HTMLEncode(EcommerceFunctions.
GetManufacturer(Eval("SKUManufacturerID"),"ManufacturerDisplayName").ToString())) %>
```

You can find other transformation examples in Writing transformations.

Summary

- Encode strings before they are rendered.
- Encode all strings from any external source (user input, database, context/environment, URL parameters, method parameters).
- Use **HTMLHelper.HTMLEncode()** to encode strings from any external source.
- For URL parameters, you can use **QueryHelper.GetText()** if that value goes directly to output (i.e. the value is not saved to the database, filesystem, etc.).
- Values from any external source rendered as a part of JavaScript code must be encoded with **ScriptHelper.GetString()**.
- In JavaScript code, never render anything from any external source directly – let the server encode these values.
- Configure cookies as http-only.
- Disable the **Enable WYSIWYG editor** option for any Forums that you have on the live site.