Kentico CMS Security White Paper

Dominik Pinter

Kentico
unlimited website possibilities

www.kentico.com

Overview

We at Kentico recognize web security as crucial aspect of our system. As we want to prepare and provide a highly secure web content management system, we prepared a document for developers, which describes writing of secure code. This paper has started as an internal guide for our developers, but we decided that it could be helpful to all web developers using Kentico CMS.

This paper intends to describe the most common security vulnerabilities we have to face in the Internet environment. You will learn how to identify them and how to secure your website against them.

The paper is focused on security of Kentico CMS. But, if you are not familiar with this web content management system, you can still use it to understand website security in general and to improve security of your websites.

Table of Contents

1. XSS

1.1. What is XSS

We all know XSS (cross site scripting), right? XSS is when you insert some evil input into a form (for example, a piece of HTML code). After that, the evil input is stored in the database. Then, on some other page, it is rendered. Browser renders attacker's evil input as a part of the page's HTML and that's it.

Well, this is definitely true. But it is just one part of the whole story. This type is called **persistent XSS**. You can find a code example of a typical situation below in the "Examples of XSS" section.

Another type of cross site scripting is called **non-persistent XSS**. Basically, it's very similar. The main difference is that a web application (like an instance of Kentico CMS) doesn't store evil input to the database in this case. Instead of storing it, the application renders the input directly as a part of the page's response. Again, you can find an example in the "Examples of XSS" section.

A special case of non-persistent XSS is called **DOM-based XSS**. This type of attack is done without sending any request to the web server. The attacker injects JavaScript code directly. Please look at the "Examples of XSS" section for better understanding of this type of attack.

1.2. Examples of XSS

1.2.1. PERSISTENT XSS

Let's 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.

```
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. The attacker can simply create a user with their first name like **"<script>alert(1)</alert>"**. Anyone who sends a request to a page where the attacker's first name would normally be shown gets the HTML code injected and executed on their machine. The user gets an alert showing **"1"**.

1.2.2. NON-PERSISTENT XSS

Let's have another standard .aspx page with a label:

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

In code behind, we have this code:

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

This is an example of a potential vulnerability to non-persistent XSS. 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)** is executed.

1.2.3. 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://localhost/55_final_security/XSSDOMBased.aspx?default=<script>alert(document.cookie)</script>, 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.

1.3. What Can Be Done by XSS

You can go to the following section if you are not interested in web application security.

Now let's imagine how the attacker can exploit our application by XSS. The question is: "Why exploitation of this vulnerability can be dangerous?". 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 change the look of your site, change its content or insert any JavaScript code to your site. With JavaScript, he can read and steal the user's cookies. With the stolen cookies, the attacker can log on to a site (even using an administrator account) without a user name and password. I personally think that this is the most dangerous thing the attacker can do when your site is XSS vulnerable.

Another example of dangerous behavior is that the attacker can redirect the page to some evil one without you even knowing it, or force you to download some evil code (typically a virus).

Okay, now we can imagine what can be done. You may say 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, maybe, 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%3e**?

Can you read that string? It's same string as before, only in hexadecimal encoding, so the same text as in the first case is sent to the browser.

1.4. How to Find an XSS Vulnerability

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 points where properties (of Kentico CMS's objects) are used and check whether they are HTML encoded when they are rendered to the page. That's the best technique of searching for persistent XSS.

Yet another way is to search for points where of context variables (**HTTPContext.XXX**, etc.) or URL parameters - **QueryHelper.GetString()** and **QueryHelper.GetText()** - are used. This way, you can find non-persistent XSS.

If you want to search for DOM-based XSS, search your code for the JavaScript objects below (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. BUT they often find far too many false positive vulnerabilities and searching using them is IMHO 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).

1.5. How to Avoid XSS

All examples were functional because 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's properly encoded? In Kentico CMS, we basically have three methods to avoid XSS:

- **HTMLHelper.HTMLEncode()** – encodes HTML tags, replaces the < and > chars with their HTML entities.
- **QueryHelper.GetText()** – gets a HTML encoded string from query string.
- **ScriptHelper.GetString()** – replaces special chars like an apostrophe (alt+39).

Another question you may have is: "Where should I protect my code?" The answer is always on output before rendering. It's because we want to secure our web, but don't want to change users' input data. We also don't want to rely on input validation.

An important note about DOM-based XSS: Any encoding on the server-side cannot protect you from this type of attack. We currently don't have any special client-side mechanism (i.e. 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 (i.e. document.write(), document.location, …) in client-side code, don't use input directly. Let the server encode the potentially evil input.

1.5.1. 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 (i.e. label text). For example:

```
lblUser.Text = 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 = QueryHelper.GetText("info", "");
```

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

```
ltScript.Text = ScriptHelper.GetScript("alert('" + dynamic + "');");
```

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

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

1.6. 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.

2. SQL Injection

2.1. What is SQL Injection

SQL injection is a well known web application vulnerability. The attacker's goal is to execute his own SQL code on the victim's database through a web application. How can the attacker do that? The attack is similar to XSS, the attacker inserts a special string into the web application via a form or a url parameter (…). If that string is used as a dynamic part of an SQL query (i.e. a part of a WHERE condition) and not protected properly, the attacker can inject a query before it's executed.

We can divide vulnerabilities into two kinds – "classic" and "blind". The only difference between them is whether the attacker can see the real error message from the SQL server (classic) or just a general error page (blind). Blind SQL injections are harder to exploit because the attacker doesn't know how exactly they can inject code.

2.2. Example of SQL Injection

We have a simple web page with a textbox and a 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 code behind, we have this code:

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

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

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

This is a correct query which doesn't cause any problems. But if a 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 and a deleted table is the result.

2.3. What Can Be Done by SQLIinjection

We already saw that with SQL injection, you can delete a table, but what else can you do? Simply, with an SQL injection vulnerability, the attacker can do exactly the same operations with the database as the web application itself. For example, you can read all data or the database schema, change it, edit it, etc. Also, T-SQL supports the xp_cmdshell() function which executes operating system commands. So, you can basically manage the whole server.

2.4. How to Find SQL Injection Vulnerabilities

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

- 'whatever – basic test
- DROP
- Something

to all inputs/url parameters/whatever. 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 (more on this topic is in the next chapter). Then, you can check variables which are inputs of these methods. The aim of checking is searching for protection against SQL injection.

You can also use automatic tools. There are two kinds of them. The first kind is based on trying – simply runs an application, tries to insert some payloads to output and checks the application's reaction. The second kind is based on searching for patterns (via regular expressions) in code. Again, these automatic tools are very inaccurate and we do not recommend to use them.

2.5. How to Avoid SQL Injection

There are many ways of protection. In Kentico CMS we are using SQL parameters and apostrophe escaping. Both methods have advantages and disadvantages. We use them in different situations.

First of all, some points about executing queries in Kentico CMS. Kentico CMS has its architecture divided into layers. One of them is the data layer and this layer provides operations for manipulation with database data. This includes executing SQL queries. You all know the GeneralConnection.ExecuteQuery() method. It is the most frequently used method for executing SQL queries in Kentico CMS. This method has many overloads. In the rest of this chapter, I will use the following overload:

```
ExecuteQuery(string queryName, object[,] parameters, string where,
string orderBy, int topN, string columns)
```

Now, let's take a look at queries with SQL parameters. The second parameter of **ExecuteQuery()** is an N-dimensional array. To fill this parameter, we use an array of three-dimensional arrays. The first dimension of the inner array is used for parameter name and the second one as its value. Third parameter is for special handling. Its exact meaning is not important for purposes of this paper. A query using these parameters could look like this:

```
INSERT INTO table1 VALUES(@param, @param2) ;
```

In code, you fill an array like this:

```
params[0, 0] = "@param" ;

params[0, 1] = "value1";

params[1, 0] = "@param" ;

params[1, 1] = "value2" ;
```

SQL server simply replaces **@param** with your value. An important fact is that the value is treated as literal. It means that even if your value contains a piece of SQL code, the SQL server doesn't execute it.

Parameters are almost 100% secure. BUT if you build a query from parameters which is executed with the built-in **exec()** function, the parameters are taken the standard way (the query is executed with them, even if they contain some evil SQL code). Look at the example procedure below. It shows a typical example of how NOT to do it.

```
CREATE PROCEDURE injection( @param varchar(30) )
AS
   SET NOCOUNT ON
```

```
    DECLARE @query VARCHAR(100)
    SET @query = 'SELECT * FROM ' + @param
    exec(@query)
GO
```

Parameters are used in Kentico CMS mainly in the INSERT, UPDATE and DELETE query types, and also in stored procedures.

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

```
searchText = searchText.Replace("'", "''");
where += " LIKE N'%" + searchText + "%'";
```

The where variable is used as the third parameter of the **ExecuteQuery()** method. This is a correct solution, but only for string values. Never use this method for other types than strings. For example:

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

In this example, replacing is useless because class Guid can only contain letters and numbers in a specific format. So, this code is secured without any protection.

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

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

The piece of code above is completely wrong. Why? There are two reasons. First of all, you don't need the apostrophe character to drop a table. In the examples above, you needed an apostrophe to inject a string constant inside an SQL command, but there are no enclosing apostrophes. You may say that the attacker cannot use WHERE conditions because you escape apostrophes – so your code is partially secured. And that's the second thing - in SQL, there is the Char() function which converts numeric values to their ASCII representations. And these ASCII letters can be concatenated so the attacker can write anything into the query.

These situations can be solved by adding enclosing characters or by converting values to correct data types. Always use the second choice (the first one causes performance issues). A correct piece of code should look like:

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

2.6. Summary

- Protect dynamic parts in INSERT, UPDATE and DELETE queries with SQL parameters.
- Don't ever use the **exec()** function in your SQL code.
- When you build a SELECT query in code, all used strings taken from external sources must be protected with **Replace("'","''")**.
- Always escape values from array(list, …) when you are getting them and putting them into a string (typically in foreach loops).
- Never rely on JavaScript validation. JavaScript is executed on the client side so the attacker can disable validation.
- When you work with other than string types, always convert data types to that type or validate the value via regular expressions.

3. Argument Injection

3.1. What is Argument Injection

Argument injection is a type of attack performed by changing input parameters of a page, enabling a user to see data which he normally cannot see. It also includes situations when the user can modify data that they would not be able to modify via the user interface, again by means of changing arguments of the page. Let's take a look at some examples for better understanding of this type of attack.

3.2. Examples of Argument Injection

Let's have a simple .aspx page which allows users to change their passwords:

```
<asp:TextBox runat="server" ID="txtPassword" >
</asp:TextBox><asp:Button runat="server" ID="btnConfirm"
onclick="btnConfirm_Click" Text="Change password" />
```

In code behind, we handle the **OnClick()** event of the button:

```
UserInfoProvider.SetPassword(QueryHelper.GetString("username",   ""),
txtPassword.Text.Trim());
```

The standard way of using this page is that the user adds a link to it in a user profile page with url parameter "username" equal to the current user's user name. In this code, there are two security issues. The first one is that the user can change their password without entering the original one. Imagine a situation when a user forgets to log out on a computer in an Internet café. And more important for us right now: is the page is vulnerable to argument injection. Any site visitor can change a password of any user of the system just by typing the URL address of the page with an appropriate user name in the parameter.

3.3. What Can Be Done by Argument Injection

You can go to the following section if you are not interested in web application security.

The answer to the question in the title is: "It depends on how your application is written". Valuable information, isn't it? Well, usually, you can read documents or view images belonging to different users and so on. This is not highly dangerous. But sometimes, you can read invoices or other kind on sensitive information. And the worst case is if you can change something. You could probably never change a user's password. But what if an application has a DeletePicture.aspx page which deletes a picture whose IDs is provided in a URL parameter…

3.4. How to Find Argument Injection

The bad thing about argument injection is that any input from an external source can cause it. And there isn't any exact way to find these bugs. You should look at every single input. But there are some practices which may help you find the most vulnerable places.

First, you should check all inputs of pages in paths containing "CMSPages". These directories contain, among others, pages which send files to the client browser and in one of the URL parameters, there is usually a path or an identifier to a file.

Second, check pages which work with IDs/names taken from querystring or via a form field. Especially those that take user IDs or site IDs from querystring. These values can be taken from context instead in many cases.

3.5. How to Avoid Argument Injection

Let's look again at the example above and for now, forget about the issue with not entering the original password. The problem is that anyone can specify any user name. We can solve this by:

- Changing the user name to an identifier which is harder to guess => a GUID.
- Taking the user name from CMSContext.
- Checking that the current user has permissions to change their password.

But what solution should we choose? The ideal solution is a combination of all of them. First, every time you do an action (displaying a picture is an action too), you must check the user's permissions. Also, if you can take data from current context, do not take if from another external source. Data in the current context, for example the information about the current user, is always correct (users can not manipulate with them). And if you have to manipulate with non-context data, use GUIDs instead of names or simple IDs.

3.6. Summary

- Always check that users have sufficient permissions to perform an action (if it's possible).
- Do not use querystring/form values if is not necessary. Instead, use CMS context values.
- If you have to use querystring/form values, use GUIDs instead of IDs or names.
- Secure querystring parameters with hash code validation.
- Combine the rules above.

4. Code (Command, …) Injection

4.1. What is Code Injection

In ASP.NET, developing a code injection is not a well known error. It's because in ASP.NET, code files are not inserted one into another dynamically (like in PHP). Programmers can only register controls in the web.config file or on a page directly. But dynamic code injection in ASP.NET is still possible. The aim is to insert C# (VB.NET, …) code that is executed directly.

4.2. Example of Code Injection

How can the attacker do such a thing? For example, if you are using the ProcessStartInfo class in your code and executing commands which are put together from external sources. Another example can be that the attacker somehow inserts a file with code into you application directory. Another possibility is if your virtual path provider is able to read files from different servers and parameters are taken from an external source. Or a situation when a developer loads a control dynamically and the source of the control is loaded from an external source.

4.3. What Can Be Done by Code Injection

The answer is simple – simply anything that can be achieved programatically.

4.4. How to Find Code Injection

There is no exact procedure to find code injection, but there are some tips for possibly vulnerable places in Kentico CMS:

- Search for **ProcessStartInfo** in source code and check its input parameters.
- Analyze the virtual path provider module and search for any possibility of getting a file which is not a regular Kentico CMS virtual file.
- Try to edit a transformation without administrator privileges.
- Search for usages of the **LoadControl()** method and check input of the method.

4.5. How to Avoid Code Injection

Most of you will never have to deal with this issue because they only pose a threat in the special cases described above. So:

- Never Load controls dynamically when their path is taken from an external source.
- Do not ever use ProcessStartInfo and other classes which execute commands or run .NET code.

- If you want to customize the virtual path provider or transformation management, be very careful.

5. XPath Injection

5.1. What is XPath Injection

The principle of XPath injection is very similar to SQL injection. The goal of the attack is very similar too. We can basically say that these attacks are the same but instead of using an SQL database, we use an XML file for data storage. One of the ways how to get data from an XML file is to use a special query language called XPath. See the following examples to understand what a vulnerable place can look like.

5.2. Example of XPath Injection

Let's say that a developer stores authentication data in an XML file with the following structure:

```
…
<user>
      <name>UserName</UserName>
      <password>Password</password>
</user>
…
```

On authentication, the developer builds an Xpath expression this way:

```
string//user[name/text()='"txtUserName.Text"' and password/text()='"
txtPassword.Text"'])
```

The txtUserName and txtPassword variables are standard ASPX textboxes. In case that a user inserts an expression with an apostrophe (') to one of the textboxes, the user terminates the string and is able to write their own XPath expression. As you can see, it's basically the same as SQL injection.

5.3. What Can Be Done by XPath Injection

You can get, modify or delete anything stored in the given XML file. Nothing more, nothing less.

5.4. How to Find XPath Injection

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

- 'whatever – basic test
- DROP
- Something

to all inputs/url parameters/whatever. If you see any error related to classes which provide manipulation with XMLs in ASP.NET, you probably found an XPath injection threat.

The second way is to search for vulnerabilities in code. You can search for the following strings:

- Xpath – many classes which work with XPath have the 'xpath' string in their name.
- **SelectSingleNode()** and **SelectNodes()** – methods used in Kentico CMS for getting data from XML files via XPath.

5.5. How to Avoid XPath Injection

These days, there is no sensitive data stored in an XML file in Kentico CMS. I also didn't find any code where an XPath expression is built the way I described in the example above. But maybe, this will change in the future. You can avoid XPath injection by following these rules:

- Validate input from external sources before you put it into XPath expressions.
- For characters like ', <, >, etc., use replace entities. "'" is a replace entity for the apostrophe.

6. Cross Site Request Forgery (CSRF/XSRF)

6.1. What is CSRF

A few years ago, Cross Site Request Forgery was not taken as a serious bug. It wasn't even taken as bug at all. But today, web is about lots of money and many non-IT users manage important websites and that's why this kind of bug is very popular these days. An important condition for a successful attack is that a user must click the attacker's link.

A browser typically uses two ways of requesting web applications – sending data via URL parameters where HTTP GET request is used, and sending data via forms where HTTP POST is used. The application typically does some action – inserts a new user into a table, deletes a forum post, etc. Nothing strange? Yes, but … but there is one problem – the web application typically doesn't check if requests are generated by the web application itself (= user clicks a link or sends a filled form). Still seems okay? Let's continue. What if the attacker creates a link for some action and sends it to the user? The user clicks the link and the action is performed without the user even noticing. And this is called Cross Site Request Forgery.

We already know that users have to click on the attacker's link or fill their form. Another condition is that the user must be logged on to the vulnerable web, but these days, almost every application provides the "keep me logged in" functionality. ASP.NET complicates a successful attack because of ViewState. If ViewState is turned on, you cannot send tampered POST requests to an ASP.NET application because validation of ViewState fails. So, many developers think that an ASP.NET application is bulletproof against CSRF. But there are always a few catches:

- GET requests can still cause CSRF.
- ViewState can be generated outside an application if you are not use machine keys as keys for ViewState encoding.
- Even if you are using machine keys to encrypt your ViewState, you are not 100% safe. ASP.NET doesn't take form values from Request.Form but from Request.Params. This is the reason why it is possible to perform something called a "One click attack". It is a special case of CSRF. You simply send ViewState and values of form fields via GET. The trick is that you can use ViewState generated by ASP.NET after post, change values of fields and validation still succeeds.

6.2. Example of CSRF

We have a simple page without any content, we only have this piece of code in code behind:

```
if (!string.IsNullOrEmpty(Request.Form["UserID"]))
    {
       DoSomeAction(Request.Form["UserID"]) ;
    }
```

Doesn't matter what the DoSomeAction() method does. The important thing is that in this case (if a machine key is not used to encode ViewState), the action is performed with the UserID specified in the UserID field. Anyone who sends a form with this field can perform the action (if he or she is authorized). And there is no checking if the user really wants to do that action.

Let's have another page without content with code behind similar to the piece of code below:

```
if (CMSContext.CurrentUser.IsGlobalAdministrator)
    {
       int userID = QueryHelper.GetInteger("UserID", 0);
       if (userID != 0)
       {
             Response.Write("I've just deleted user with id: " +
             userID);
       }
    }
else
    {
       Response.Write("You don't have enough permissions to delete
       user");
    }
```

This piece of code is similar to the code above. The only difference is that now, UserID is taken from querystring (by the GET method).

The third example shows a one-click attack. Let's have a simple page with a textbox and a button. The code below handles the Onclick action of the button:

```
protected void btnSend_Click(object sender, EventArgs e)
    {
       Response.Write(txtUserID.Text);
    }
```

Users typically insert a value into the txtUserID textbox and click the button. But the attacker can forge a link similar to this one to the user:

http://site/Page.aspx?__VIEWSTATE=/wEPDwULLTE0MDM4MzYxMjNkZIb5PxpCoDI4Dt3C2LKzz8CnHkbd&txtUserID=<anything>&btnSend=Send&__EVENTVALIDATION=/wEWAwLdr4fPBgLT8dy8BQKFzrr8AbhBL27NfMMamif/pHIFUlo41HNI

The **<anything>** macro can be changed to anything else by the attacker. ViewState is taken from the page that can be generated after postback on that page. Validation is successful.

6.3. What Can Be Done by CSRF

It depends on how badly an application is written. If it is very bad and the administrator of the web doesn't take care of the server properly (for example, encoding of ViewState based on machine key being turned off), then the attacker can do anything that the victim of the attack could normally do.

6.4. How to Find CSRF

First, how to find a GET CSRF bug. If you find any page/control/etc that does an action on GET request, there is a possibility of a CSRF bug. For example, try to find the following strings in your source code:

- QueryHelper.GetString("action")

Try some other similar strings. You can also search for strings:

- EnableViewState="false"
- EnableViewStateMac="false"

If you find these strings in the **<%@ page** directive, it means that a developer turned off ViewState validation (first case) or machine keys for ViewState encoding (second case). You already know that ViewState validation helps a lot to avoid POST CSRF. So globally, it must always be turned on.

Also, if you find any page that does not inherit from our class, it means that there is a possibility of CSRF.

Of course, if a page doesn't have these features and the page doesn't do any actions, so it doesn't have to be protected from CSRF.

6.5. How to Avoid CSRF

If you have read the section above, you probably have ideas for what you should do. First, it is written in the intro to CSRF that even if your application encodes ViewState properly, a

special case of CSRF, a one click attack, is still possible. The problem is simple – ViewState is the same for all users. However, you can specify a key corresponding to the current user by the ViewStateUserKey page property. The recommended value is a user's session ID. All CMS pages must inherit from CMSAbstractPage (the base page for all CMS pages). CMSAbstractPage page sets ViewStateUserKey to a unique value for every user => avoids one click attacks.

Because ASP.NET partially secures web applications from POST CSRF and we add the least needed functionality to protect all POST requests by default, always use POST requests only for actions.

To enable machine key encoding or ViewState validation, you don't need to do any actions.

By default (on the level of the global web.config file), ASP.NET is set to:

```
<machineKey validationKey="AutoGenerate,IsolateApps"
decryptionKey="AutoGenerate,IsolateApps" validation="SHA1"
decryption="Auto" compatibilityMode="Framework20SP1" />

<pages buffer="true" enableSessionState="true"
enableViewState="true" enableViewStateMac="true"
viewStateEncryptionMode="Auto"
```

It means that machineKey is generated automatically, ViewState is validated and encoding of ViewState based on machineKeys is also enabled. If a control requests it, ViewState is encrypted by SHA1.

6.6. Summary

- Do not use GET requests to perform actions, always use POST.
- Never turn off validation of ViewState on a page (key EnableViewState) globally.
- Never turn off encoding of ViewState based on machineKey (EnableViewStateMac) on page globally.
- Do not set the CMSUseViewStateUserKey key to false (it is an internal key which can cause insertion of the current user's key to ViewState encoding).
- If you insert a new page, always make it inherit from some of the CMS pages.
- If you create a new CMS page class, check that your page directly or indirectly inherits from CMSAbstractPage.

7. Session Attacks

7.1. What Is a Session Attack?

In this part of the paper, we are starting with OWASP's top 3 bugs/issues for the year 2010. There are basically three types of session attacks – session stealing, session prediction and something called "session fixation". Let's talk about them in the preceding order.

But first, let's remind a few facts about sessions. Web is running on HTTP, which is a stateless protocol. But in many web applications, we need some state information, context if you want. This is the purpose of sessions. When a user opens their browser a goes to some website, the web server of that website generates a session ID for them. The session ID is sent with every request and it is a key for session data (session data = state/context). These are stored on the server. Session IT can be given to the request in two ways – via a URL parameter or by a cookie. The session ends (in a typical case) after the user closes their browser or after a specified amount of time of the user's inactivity.

We talked about session stealing in Part 1 of Kentico CMS Security Paper. Do you remember where? Yes, in the part where XSS was introduced. XSS is probably the most popular method for session stealing. Why? And how can be a session stolen? When I say session stealing, I mean stealing of a session ID. In both cases, these IDs can be read by JavaScript. If you know the ID, you can exchange your ID with the stolen one and all session data belong to you.

Why does the attacker try to steal session IDs? Why doesn't he just guess some random session ID? Most of the implementations of session IDs are long strings and guessing a correct session ID in linear time is impossible. But there are also bad implementations when the attacker can generate session IDs from known values. And this technique is called session prediction. For example, imagine that a session ID is a user name encoded in base64. FYI: on the last DefCon (number 18 in 2010), Samy Kamkar showed how to predict session IDs in PHP (version 5.3.1 – not a historical one). Fortunately, in ASP.NET, session ID is a 120bit random number represented by a 20-character string. So, it's relatively safe.

The last type of attack is called session fixation. In this case, the attacker lets the server generate the right session ID for them. Then, they must tamper this ID to a regular user. This situation is quite easy when session id is given in url parameter. After tampering, the user and the attacker are sharing the same session. For example, when a user gets authenticated, the attacker is authenticated as a user too.

The goal of all kinds of session attacks is the same – get user's session data or identity forgery. Because of this fact, in the rest of the "session attacks" issue, I will aim on Session

fixation. Session fixation is also just another type of attack where something new can be shown (you have already seen XSS and we simply cannot influence how session IDs are generated).

7.2. Example of Session Fixation

Let's have a simple .aspx page which saves a value to a session and also shows it:

```
<asp:Literal runat="server" ID="ltlSession"></asp:Literal>
<asp:TextBox runat="server" ID="txtValue"></asp:TextBox>
<asp:Button runat="server" ID="btnSend" Text="Save" />
```

In code behind, we handle the OnClick() event of the button:

```
Session["MyPrivateData"] = txtValue.Text;
```

On page load, we display the value via the literal:

```
ltlSession.Text = "My private Data:" + SessionHelper.GetValue("MyPrivateData") + "<br/>";
```

In this case, a user can save some private data and the attacker can see it. Now, think about how the process of authentication in a typical case works. The user fills their credentials into a form, these are verified and then a flag or user info is set to the session. The attacker probably cannot see the infos in this case, but they are authenticated with the user credentials.

7.3. What Can Be Done by Session Fixation

As I wrote above, the main goal is to read/manipulate with session data or identity forgery. In both cases, it depends on the particular application how much damage can be done. If the application stores sensitive data to sessions (for example user passwords in plain text) and allows to show these or allows to change them, the damage can be really huge.

7.4. How to Find Session Fixation

Until version 6.0, session fixation is possible in Kentico CMS and it can be still possible in 6.0 (it will depend on application settings). But we have to ensure that we will not store any sensitive information in sessions. The best way is to determine which variables are stored in sessions. Then, check how you can manipulate with them (read/change) and think about the risk if the attacker manipulates with them. You can find these just from the user perspective by looking at application reactions, parameters and so on, but I recommend code inspection.

You can simply find all manipulations with session data by searching for SessionHelper and the Session[] array.

7.5. How to Avoid Session Fixation

First of all, there is no native support for avoiding session fixation in ASP.NET. A few years ago, there was a request for Microsoft to solve session fixation. But Microsoft said that is up to developers. Microsoft arguments that session management and authentication are two separate modules and the attacker must tamper an authentication cookie, which is quite difficult. They are partially right, but many applications (including Kentico CMS) use sessions for storing current user info and other user-related data. That's the reason why developers think that protecting the authentication cookie is not enough. The best practice for protecting your application against session fixation is to regenerate the session ID after a user logs on. You must do it by yourself in your code. The best way to do it is to change the session ID to an empty string and let ASP.NET generate a new one. However, by this action, you lose your session data.

In Kentico CMS, we don't do anything special for protection from session fixation. We have introduced the "Impersonation" functionality which allows one user to log on as another one. By implementing this functionality, we have basically secured Kentico CMS from session fixation because it tests if the authenticated user (taken from HttpContext – authentication cookie) is the same as the user in the session. If not, the user info in the session is changed to the right one. But it is still possible to manipulate other data. All you need to take care of is not to save sensitive/critical information to sessions.

FYI: We are planning to add functionality for regenerating session IDs after a user logs on in version 6.0. This feature will be, however, disabled by default because impersonation will not work with this feature enabled.

8. Directory Traversal

8.1. What is Directory Traversal

This type of attack is also known as path traversal. The main goal here is to show content of a file or directory via an application. Applications read data from the file system in many cases. Paths to these files/directories are often taken from input. If a user's input isn't handled carefully, users can read data from the root directory of the server's file system.

8.2. Example of Directory Traversal

Look at the following code:

```
Response.Write("<strong>Absolute path e:\\: </strong><br />");
string[] dirs = Directory.GetDirectories("e:\\") ;
foreach (string dir in dirs)
        {
            Response.Write(dir + "<br />");
        }

Response.Write("<br /><strong>Read e:\\StorageHelper.cs with
absolute path: </strong><br />");
Response.Write(File.ReadAllText("e:\\StorageHelper.cs") + "<br />");

Response.Write("<br /><strong>Root path of C: taken by ./../../../:
</strong><br />");
string[] dirs2 = Directory.GetDirectories("./../../../");
foreach (string dir in dirs2)
        {
            Response.Write(dir + "<br />");
        }

Response.Write("<br /><strong>Read c:\\StorageHelper.cs with
./../../../: </strong><br />");

Response.Write(File.ReadAllText("./../../../StorageHelper.cs") +
"<br />");

Response.Write("<br /><strong>Directory with application: taken by
ServerMapPath(\"~/../\"): </strong><br />");
string[] dirs3 = Directory.GetDirectories(Server.MapPath("~/../"));
foreach (string dir in dirs3)
        {
            Response.Write(dir + "<br />");
        }
```

```
//This code throws the exception - but poor security here
Response.Write("Directory with application: taken by /../: <br/>");
string[] dirs4 = Directory.GetDirectories(Server.MapPath("/../."));
foreach (string dir in dirs4)
`      {
          Response.Write(dir + "<br />");
       }

Response.Write("<br /><strong>Directory with application: taken by
Server.MapPath(\"/\") + \"../: </strong><br />");
string[] dirs5 = Directory.GetDirectories(Server.MapPath("/") +
"../");
foreach (string dir in dirs5)
        {
            Response.Write(dir + "<br />");
        }
```

The result of this code will look like this:

```
Absolute path e:\:
e:\$RECYCLE.BIN
e:\ASP.NET_TEMP
e:\AzureSLN

Read e:\StorageHelper.cs with absolute path:
************ Some code 2 ************

Root path of C: taken by ./../../../:
./../../../$Recycle.Bin
./../../../Backup
./../../../Boot

Read c:\StorageHelper.cs with ./../../../:
************************** Some code StorageHelper.cs
**************************

Directory with application: taken by ServerMapPath("~/../"):
C:\inetpub\wwwroot\3855_10914
C:\inetpub\wwwroot\3889_32518
C:\inetpub\wwwroot\3968_07397

Directory with application: taken by Server.MapPath("/") + "../:
C:\inetpub\wwwroot\../AdminScripts
C:\inetpub\wwwroot\../custerr
C:\inetpub\wwwroot\../history
```

These are listings from my hard disk. The exact paths or listings aren't important, the important thing is that you can simply read any file or list any directory. And you can either use a relative or an absolute path. The situation here can make an easier (from the attacker side) vulnerability called full path disclosure. This is a vulnerability or bug when an error message contains a full path. The attacker finds out your directory structure and can effectively exploit a directory traversal bug.

8.3. What Can Be Done by Directory Traversal

The first dangerous thing is that the attacker knows your directory structure. And if you have some sensitive information in files in your application directory, they can simply download these files. A worse case is when your application enables the attacker to read files. The attacker can read your configuration files (e.g. the connection string in web.config) or the configuration files of the whole system (if the application is used by a highly privileged user).

8.4. How to Find Directory Traversal

Search for parts of your application where the application reads files/directories. Then try to change input parameters to get content from a place outside the application directory. In code, you can search for these strings:

- ServerMapPath
- FileStream
- StreamReader

8.5. How to Avoid Directory Traversal

Validate user inputs, check for absolute paths, for sequences of "../" and so on. Also, if you read a file, check if the file is within the application path, the application should never read data from a random path on a hard disk.

9. Unvalidated Redirects and Forwards

9.1. What is Unvalidated Redirect

In a web application, there are typical situations where a user is redirected from one page to another. For example, after a user logs in, they are redirected to the main page. Where is the catch? If the target of redirection is inserted as a parameter into the URL of the login page, it can be changed. The problem is that the attacker could prepare a login link similar to this one:
http://www.mysupersecurepage.tld/login.aspx?redirect=http://hackerspage.tld. This link will be posted to a forum at mysecurepage.tld. After that, a user clicks the link, logs in to the regular page and then they are redirected to the attacker's page. And that's it.

9.2. Example of Unvalidated Redirect

Look at the following code:

```
UrlHelper.Redirect(Server.UrlDecode(QueryHelper.GetString("redirect"
, "")));
```

This code could be executed after the logon procedure. You can see that there is no protection from redirection to a foreign site. But why should the site redirect the user to another one?

9.3. What Can Be Done by Unvalidated Redirect

The answer is simple – redirect the user to another site. This could be dangerous when the user does not recognize that they are now on a different site. The attack scenario can look similar to this one: the user logs in, they are redirected to hacker's site which looks exactly the same as the previous one, but in the address bar, the user now sees an IP address (but where's the problem – a few years ago, Google cache ran on IP addresses too). So, the user is on an evil site and sees a message that they have entered a wrong user name or password (while they haven't). The user says "What the hell?" and inputs their credentials once again. Now everything is OK, the user is redirected back to the regular site where they are successfully logged on, but the attacker has their credentials.

9.4. How to Find Unvalidated Redirect

Search for redirections in your application, you can search for bugs from the user point of view, but I strongly recommend searching for the following strings in your code:

- UrlsHelper.Redirect
- Location

9.5. How to Avoid Unvalidated Redirect

You have two options here. First, provide the redirection target in a secure way, for example, secure the parameters by a hash. Second, validate if the target of redirection is a part of your application or not. The protection should look similar to this piece of code:

```
url.StartsWith("~") || url.StartsWith("/") ||
QueryHelper.ValidateHash("hash")
```

10. DoS on Application Layer

10.1. What Is DoS on Application Layer

DoS is an abbreviation for “Denial of service”. The goal of this type of attacks is to “kill” or overload the system/application/whatever so that it cannot respond to users. DoS always originates in a weakness or bug in implementation. There is yet another kind of attack called DDoS = distributed DoS. This kind of attack typically doesn’t need a bug to success because it uses brute force of botnets. The attacker simply sends millions of requests to a service in one time.

But today’s topic is the DoS on application layer. So we want to kill a part or the whole application, web application in our case. A good target is a database because almost every web application depends on a database. For this purpose, SQL wildcards are a very useful weapon.

Another good example is flood protection. Flood protection is a good protection mechanism against SPAM, but only when it’s implemented right. If it’s not, it can be used by the attacker to cause DoS.

10.2. Example of DoS on Application Layer

Let’s look closer at SQL based DoS. When you have some search box in your application, and ordinary user searches for strings like “apple” or “microsoft”. But the attacker can insert a string similar to this one:

```
%_[aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaa%aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa%aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa%aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaa%aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa%aaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa]_[^!_%/%a?F%_D
)_(F%)_%([)({}%){()}L$&N%_)$*L()$*R"_)][%](%[x])%a][$*"L$-9]_%
```

I searched this string in the CMS_PageTemplate.PageTemplate_WebParts column and it took 50% of our SQL server resources for about 30 seconds.

10.3. How to Find DoS on Application Layer

In case of SQL DoS, search for filters which have textboxes with maximal length. In case of other DoSes, try security features like flood protection. Check if your "bad behavior" influences other users. Try to log on to forms, post messages to forums, etc.

10.4. How to Avoid DoS on Application Layer

For SQL, you need to do two things. First, set the maximal length of filter textboxes where possible. Second, escape wildcard strings where possible.