Argument injection is a type of attack based on tampering with input parameters of a page. This can enable attackers to see data which they normally cannot see or to modify data which they normally cannot modify, via the user interface.

Example of argument injection

In this example, we 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:

```
using CMS.Membership;
using CMS.Helpers;

...

protected void btnConfirm_Click(object sender, EventArgs e)
{
        UserInfoProvider.SetPassword(QueryHelper.GetString("username", ""),
txtPassword.Text.Trim());
}
```

The standard way of using this page is that the user adds a link to this page on 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 user can change the password without entering the original one. If the user forgets to log out of a computer in an Internet café, then anybody can change the user's password.
- 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.

What can argument injection attack do

Argument injection can usually be used to obtain various information. The attacker can, for example, read pages or view images belonging to different users and so on. The threat usually depends on the sensitivity of the information. 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?

Finding argument injection vulnerabilities

The problem with argument injection is that any input from an external source can cause it. And there is no exact way of finding these spots. You should examine every single input. However, there are some practices which may help you find the most vulnerable places:

- 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.
- Check pages which work with IDs/names taken from the URL query string or via a form field. Especially those that take user IDs or site IDs from query string parameters. These values can usually be taken from the application's context instead.

Avoiding argument injection in Kentico

Let's get back to the provided example and consider the problem 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 MembershipContext.
- Checking that the current user has permissions to change his password.

But what solution is the best? The ideal solution is to combine all of them. 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.

Summary

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