SharePoint Development Security Best Practices

# Contents

[1 Contents 2](#_Toc508179069)

[2 Introduction 3](#_Toc508179070)

[3 Avoid Loading and executing code of unknown origins 4](#_Toc508179071)

[3.1 Code Access Security Policy 5](#_Toc508179072)

[4 Safe Control Entries 6](#_Toc508179073)

[5 Input Validation to Prevent Cross-Site Scripting 7](#_Toc508179074)

[5.1 Encode Output Properly 8](#_Toc508179075)

[5.2 Set Charset in the Content-Type HTTP Response Header 9](#_Toc508179076)

[6 Prevent Cross-Site Request Forgery 10](#_Toc508179077)

[6.1 Validate the Form Digest Canary Before Processing a Postback 11](#_Toc508179078)

[7 Prevent SQL Injection 12](#_Toc508179079)

# Introduction

The goal of this document is to provide a list of security best-practice recommendations to consider when you are developing SharePoint application.

# Avoid Loading and executing code of unknown origins

When designing and writing your code, you need to protect and limit the access that code has to resources, especially when using or invoking code of unknown origin. So, keep in mind the following techniques to ensure your code is secure:

* Do not use Code Access Security (CAS).
* Do not use partial trusted code.
* Do not use .NET Remoting.
* Do not use Distributed Component Object Model (DCOM).
* Do not use binary formatters.

Code Access Security and Security-Transparent Code will not be supported as a security boundary with partially trusted code.   
We advise against loading and executing code of unknown origins without putting alternative security measures in place. The alternative security measures are:

* Virtualization
* AppContainers
* Operating system (OS) users and permissions
* Hyper-V containers

## Code Access Security Policy

Code Access Security (CAS) is a resource-constraints policy that limits the access that an assembly has to protected system resources and operations.

SharePoint Foundation has built-in security policies built on top of the built-in security policies of ASP.NET. By default, SharePoint Foundation uses a minimal set of permissions in order to protect the server and underlying infrastructure from malicious code.

If your Web Part needs greater access than what is provided in the minimal settings, you can create a custom CAS policy for your Web Part, or increase the overall trust level of the server farm in the web.config file. **This is a security risk and is not recommended**.

# Safe Control Entries

SharePoint provides a security mechanism, called safe control entries, to limit the access of untrusted users to certain controls.

By design, SharePoint allows untrusted users to upload and create ASPX pages on the SharePoint server, to prevent these users from adding unsafe code to ASPX pages, SharePoint limits their access Safe Control Entries.

Every SharePoint project item in Visual Studio has a property called **Safe Control Entries** that has two Boolean sub-properties:

* Safe:

Specifies whether untrusted users can access a control

* Safe Against Script:

To help protect SharePoint sites against script injection, which is the insertion of potentially malicious code into controls or Web pages.

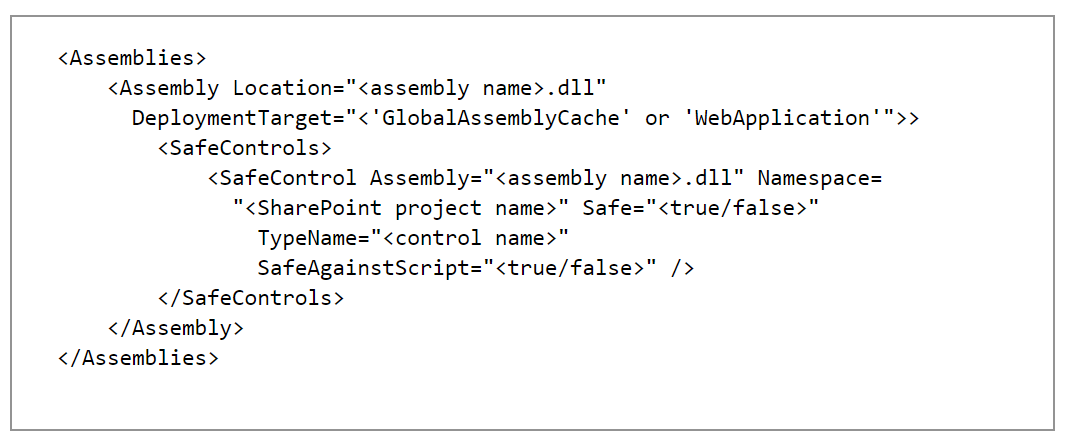
This behavior is controlled by a SafeControl attribute called SafeAgainstScript.

In Visual Studio, set this attribute in a project item's Safe Control Entries subproperty Safe Against Script.

Safe control entries are referenced on an assembly basis, you add safe control entries to a project's assembly by entering them in the project item's **Safe Control Entries** property.

Also you can add safe control entries to a project's assembly through the **Advanced** tab in the **Package Designer** when you add an additional assembly to the package.

When you add a safe control entry to a project item or to the project's assembly, a reference is written to the package manifest in the following format:



# Input Validation to Prevent Cross-Site Scripting

Input validation is performed to ensure only properly formed data is entering the workflow in an information system, preventing malformed data from persisting in the database and triggering malfunction of various downstream components.

Input validation should happen as early as possible in the data flow, preferably as soon as the data is received from the external party.

## Encode Output Properly

Encode rendered output properly using SPHTTPUTILITY methods, as attackers want to run their ECMAScript (JavaScript, JScript) script on your website because this enables them to steal authentication cookies and lure administrators into performing malicious actions.

These attackers look for places where user input is not encoded correctly, which allows the user input to be interpreted as HTML or JavaScript.

For example, a list item that is tagged <script>alert()</script> would run JavaScript if it is not HTML-encoded by using **Microsoft.SharePoint.Utilities.SPHttpUtility.HtmlEncode** before being rendered on the client.

**Recommendation**:

Before rendering user data on the client, encode it by using the appropriate method from the [SPHttpUtility](https://msdn.microsoft.com/en-us/library/office/microsoft.sharepoint.utilities.sphttputility(v=office.15).aspx) class. Sources of user data include the following:

* Data from the SharePoint object model (for example, field values and list titles)
* Parameters from the query string, headers, cookies, or form body in the request
* Web service parameters

The encoding method you should use depends on how you are using the data on the client, as shown in this table:

|  |  |
| --- | --- |
| **Type** | **Microsoft.SharePoint.Utilities.SPHttpUtility class method to use** |
| HTML tag inner text | **HtmlEncode** |
| HTML tag inner text, allow basic formatting | **HtmlEncodeAllowSimpleTextFormatting** |
| HTML tag attribute, non-URL | **HtmlEncode** |
| HTML tag attribute, URL | **HtmlUrlAttributeEncode** |
| JavaScript | **EcmaScriptStringLiteralEncode** |
| Known safe values (for example, GUID from a query parameter) | **NoEncode** |

## Set Charset in the Content-Type HTTP Response Header

Developers often defend against XSS by blocking input that contains special characters such as angle brackets < and >. Unfortunately, attackers do not need to use angle brackets to inject a <script> tag if they can trick the browser into interpreting the page as UTF-7 or another charset.

**Recommendation**:

Always specify a charset in the Content-Type HTTP response header. Usually, this should be UTF-8 format, as shown in the following example.



This specification of a charset prevents an attacker from controlling the browser's charset from within the content of the page.

# Prevent Cross-Site Request Forgery

Cross-site request forgery (CSRF or XSRF) is an attack that tricks the victim's browser into performing an unwanted action on the victim's behalf. For example, this type of attack could result in transferring funds, changing a password, or purchasing an item.

## Validate the Form Digest Canary Before Processing a Postback

If you browse to a page owned by the attacker, he could perform operations on SharePoint using your credentials.

For example, the attacker might control a page at http://contoso123. When you browse to his page, it posts to http://wingtip/\_layouts/deleteweb.aspx using your credentials. If you have administrator's permissions on http://wingtip, this would delete the http://wingtip website.

**Recommendation:**

SharePoint uses a dynamic canary to ensure that **POST** requests come from the same domain as the server.

You must do two things:

* Send the canary with every postback or web service request.
* Validate the canary before acting on the postback or web service request.

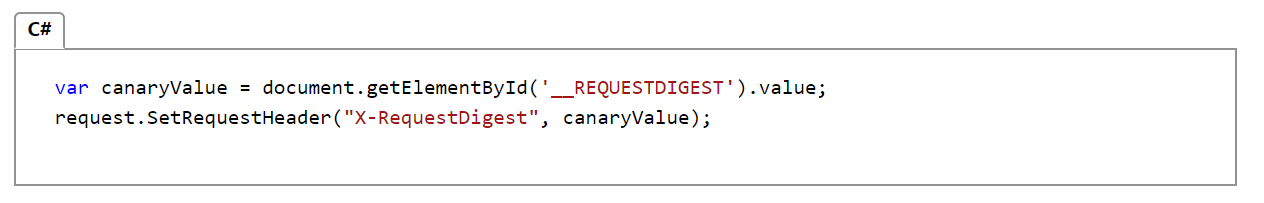
The following are the steps to take,

1. **Send the canary with every postback or web service request.**

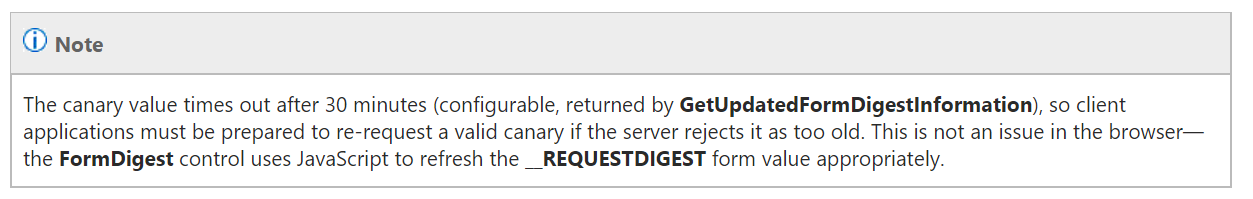
The canary value is already present in a hidden **\_\_REQUESTDIGEST** form element on every page that uses the SharePoint master page. It is automatically sent with every postback.

If you do not inherit from the SharePoint master page, you must use the [Microsoft.SharePoint.WebControls.FormDigest](https://msdn.microsoft.com/en-us/library/microsoft.sharepoint.webcontrols.formdigest.aspx) control to write the value into the page.

If you are making a web service call, you must get the **\_\_REQUESTDIGEST** value and include it in the **X-RequestDigest** HTTP request header, as shown in the following example.



Client applications (outside the browser) must manually fetch the canary value by calling the [GetUpdatedFormDigestInformation](https://msdn.microsoft.com/en-us/library/ff407486.aspx) web method on the Sites.asmx SOAP web service. This value should be included in the **X-RequestDigest** header of any web service call.



1. **Validate the canary before acting on the postback or web service request.**

Before performing any action as a result of a postback or a web service call, you must call [ValidateFormDigest()](https://msdn.microsoft.com/en-us/library/office/microsoft.sharepoint.utilities.sputility.validateformdigest(v=office.14).aspx) to verify that the canary is valid. It throws an exception if the canary is not valid.

# Prevent SQL Injection

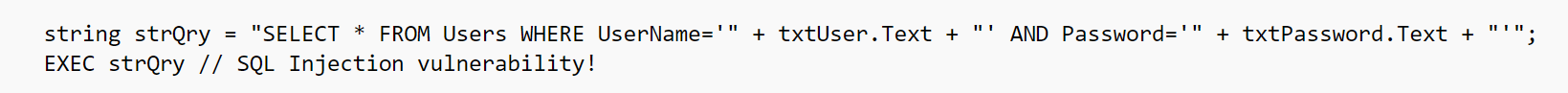
SQL injection attack exploits vulnerabilities in input validation to run arbitrary commands in the database.

It can occur when your application uses input to construct dynamic SQL statements to access the database.

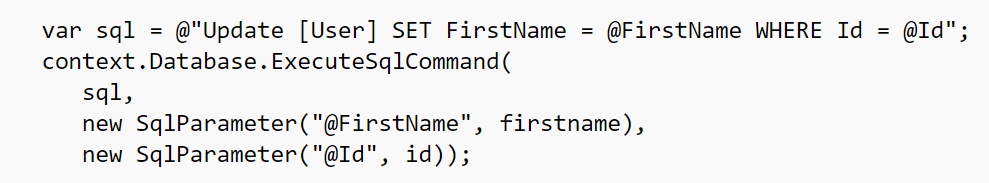
The issue is magnified if the application uses an over-privileged account to connect to the database, in this case it is possible to use the database server to run operating system commands and potentially compromise other servers.

**Recommendation:**

* All database interaction must be abstracted through stored procedures.
* No stored procedure should have dynamic SQL (concatenate strings anywhere in your code and execute them against your database).



* Applications should have no access to table or view object.
* All database calls should be parameterized instead of being inline dynamic SQL, as per the following example:



* Thoroughly validate the supplied inputs, use regular expressions to make sure they do not include SQL characters.
* Use a login to the database that is restricted and least privileged.