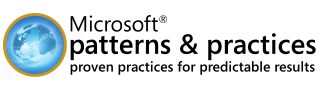
**How To: Prevent Cross-Site Scripting in ASP.NET**

21 out of 65 rated this helpful - [Rate this topic](http://msdn.microsoft.com/en-us/library/ff649310.aspx#feedback)

|  |
| --- |
| **Retired Content** |
| This content is outdated and is no longer being maintained. It is provided as a courtesy for individuals who are still using these technologies. This page may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist. |

[](http://www.microsoft.com/practices)

[patterns & practices Developer Center](http://msdn.microsoft.com/practices)

J.D. Meier, Alex Mackman, Blaine Wastell, Prashant Bansode, Andy Wigley

Microsoft Corporation

May 2005

**Applies To**

* ASP.NET version 1.1
* ASP.NET version 2.0

**Summary**

This How To shows how you can help protect your ASP.NET applications from cross-site scripting attacks by using proper input validation techniques and by encoding the output. It also describes a number of other protection mechanisms that you can use in addition to these two main countermeasures.

Cross-site scripting (XSS) attacks exploit vulnerabilities in Web page validation by injecting client-side script code. Common vulnerabilities that make your Web applications susceptible to cross-site scripting attacks include failing to properly validate input, failing to encode output, and trusting the data retrieved from a shared database. To protect your application against cross-site scripting attacks, assume that all input is malicious. Constrain and validate all input. Encode all output that could, potentially, include HTML characters. This includes data read from files and databases.

**Contents**

[Objectives](http://msdn.microsoft.com/en-us/library/ff649310.aspx#paght000004_objectives)  
[Overview](http://msdn.microsoft.com/en-us/library/ff649310.aspx#paght000004_overview)  
[Summary of Steps](http://msdn.microsoft.com/en-us/library/ff649310.aspx#paght000004_summaryofsteps)  
[Step 1. Check That ASP.NET Request Validation Is Enabled](http://msdn.microsoft.com/en-us/library/ff649310.aspx#paght000004_step1)  
[Step 2. Review ASP.NET Code That Generates HTML Output](http://msdn.microsoft.com/en-us/library/ff649310.aspx#paght000004_step2)  
[Step 3. Determine Whether HTML Output Includes Input Parameters](http://msdn.microsoft.com/en-us/library/ff649310.aspx#paght000004_step3)  
[Step 4. Review Potentially Dangerous HTML Tags and Attributes](http://msdn.microsoft.com/en-us/library/ff649310.aspx#paght000004_step4)  
[Step 5. Evaluate Countermeasures](http://msdn.microsoft.com/en-us/library/ff649310.aspx#paght000004_step5)  
[Additional Considerations](http://msdn.microsoft.com/en-us/library/ff649310.aspx#paght000004_additionalconsiderations)  
[Additional Resources](http://msdn.microsoft.com/en-us/library/ff649310.aspx#paght000004_additionalresources)

**Objectives**

* Understand the common cross-site scripting vulnerabilities in Web page validation.
* Apply countermeasures for cross-site scripting attacks.
* Constrain input by using regular expressions, type checks, and ASP.NET validator controls.
* Constrain output to ensure the browser does not execute HTML tags that contain script code.
* Review potentially dangerous HTML tags and attributes and evaluate countermeasures.

**Overview**

Cross-site scripting attacks exploit vulnerabilities in Web page validation by injecting client-side script code. The script code embeds itself in response data, which is sent back to an unsuspecting user. The user's browser then runs the script code. Because the browser downloads the script code from a trusted site, the browser has no way of recognizing that the code is not legitimate, and Microsoft Internet Explorer security zones provide no defense. Cross-site scripting attacks also work over HTTP and HTTPS (SSL) connections.

One of the most serious examples of a cross-site scripting attack occurs when an attacker writes script to retrieve the authentication cookie that provides access to a trusted site and then posts the cookie to a Web address known to the attacker. This enables the attacker to spoof the legitimate user's identity and gain illicit access to the Web site.

Common vulnerabilities that make your Web application susceptible to cross-site scripting attacks include:

* Failing to constrain and validate input.
* Failing to encode output.
* Trusting data retrieved from a shared database.

**Guidelines**

The two most important countermeasures to prevent cross-site scripting attacks are to:

* **Constrain input**.
* **Encode output**.

**Constrain Input**

Start by assuming that all input is malicious. Validate input type, length, format, and range.

* To constrain input supplied through server controls, use ASP.NET validator controls such as **RegularExpressionValidator** and **RangeValidator**.
* To constrain input supplied through client-side HTML input controls or input from other sources such as query strings or cookies, use the **System.Text.RegularExpressions.Regex** class in your server-side code to check for expected using regular expressions.
* To validate types such as integers, doubles, dates, and currency amounts, convert the input data to the equivalent .NET Framework data type and handle any resulting conversion errors.

For more information about and examples of how to constrain input, see [How To: Protect From Injection Attacks in ASP.NET](http://msdn.microsoft.com/en-us/library/bb355989.aspx).

**Encode Output**

Use the **HttpUtility.HtmlEncode** method to encode output if it contains input from the user or from other sources such as databases. **HtmlEncode** replaces characters that have special meaning in HTML-to-HTML variables that represent those characters. For example, < is replaced with **&lt;** and **"** is replaced with **&quot;**. Encoded data does not cause the browser to execute code. Instead, the data is rendered as harmless HTML.

Similarly, use **HttpUtility.UrlEncode** to encode output URLs if they are constructed from input.

**Summary of Steps**

To prevent cross-site scripting, perform the following steps:

* **Step 1. Check that ASP.NET request validation is enabled.**
* **Step 2. Review ASP.NET code that generates HTML output.**
* **Step 3. Determine whether HTML output includes input parameters.**
* **Step 4. Review potentially dangerous HTML tags and attributes.**
* **Step 5. Evaluate countermeasures.**

**Step 1. Check That ASP.NET Request Validation Is Enabled**

By default, request validation is enabled in Machine.config. Verify that request validation is currently enabled in your server's Machine.config file and that your application does not override this setting in its Web.config file. Check that **validateRequest** is set to **true** as shown in the following code example.

<system.web>

<pages buffer="true" validateRequest="true" />

</system.web>

You can disable request validation on a page-by-page basis. Check that your pages do not disable this feature unless necessary. For example, you may need to disable this feature for a page if it contains a free-format, rich-text entry field designed to accept a range of HTML characters as input. For more information about how to safely handle this type of page, see [Step 5. Evaluate Countermeasures](http://msdn.microsoft.com/en-us/library/ff649310.aspx#paght000004_step5).

**To test that ASP.NET request validation is enabled**

1. Create an ASP.NET page that disables request validation. To do this, set **ValidateRequest="false"**, as shown in the following code example.
2. <%@ Page Language="C#" ValidateRequest="false" %>
3. <html>
4. <script runat="server">
5. void btnSubmit\_Click(Object sender, EventArgs e)
6. {
7. // If ValidateRequest is false, then 'hello' is displayed
8. // If ValidateRequest is true, then ASP.NET returns an exception
9. Response.Write(txtString.Text);
10. }
11. </script>
12. <body>
13. <form id="form1" runat="server">
14. <asp:TextBox id="txtString" runat="server"
15. Text="<script>alert('hello');</script>" />
16. <asp:Button id="btnSubmit" runat="server"
17. OnClick="btnSubmit\_Click"
18. Text="Submit" />
19. </form>
20. </body>
21. </html>

1. Run the page. It displays **Hello** in a message box because the script in **txtString** is passed through and rendered as client-side script in your browser.
2. Set **ValidateRequest="true"** or remove the **ValidateRequest** page attribute and browse to the page again. Verify that the following error message is displayed.
3. A potentially dangerous Request.Form value was detected from the client (txtString="<script>alert('hello...").

This indicates that ASP.NET request validation is active and has rejected the input because it includes potentially dangerous HTML characters.

**Note** Do not rely on ASP.NET request validation. Treat it as an extra precautionary measure in addition to your own input validation.

**Step 2. Review ASP.NET Code That Generates HTML Output**

ASP.NET writes HTML as output in two ways, as shown in the following code examples.

Response.Write

<% =

Search your pages to locate where HTML and URL output is returned to the client.

**Step 3. Determine Whether HTML Output Includes Input Parameters**

Analyze your design and your page code to determine whether the output includes any input parameters. These parameters can come from a variety of sources. The following list includes common input sources:

* **Form fields**, such as the following.
* Response.Write(name.Text);
* Response.Write(Request.Form["name"]);
* Query Strings
* Response.Write(Request.QueryString["name"]);

* **Query strings**, such as the following:
* Response.Write(Request.QueryString["username"]);

* **Databases and data access methods**, such as the following:
* SqlDataReader reader = cmd.ExecuteReader();
* Response.Write(reader.GetString(1));

Be particularly careful with data read from a database if it is shared by other applications.

* **Cookie collection**, such as the following:
* Response.Write(
* Request.Cookies["name"].Values["name"]);

* **Session and application variables**, such as the following:
* Response.Write(Session["name"]);
* Response.Write(Application["name"]);

In addition to source code analysis, you can also perform a simple test by typing text such as "**XYZ**" in form fields and testing the output. If the browser displays "**XYZ**" or if you see "**XYZ**" when you view the source of the HTML, your Web application is vulnerable to cross-site scripting.

To see something more dynamic, inject <**script**>**alert('hello');**<**/script**> through an input field. This technique might not work in all cases because it depends on how the input is used to generate the output.

**Step 4. Review Potentially Dangerous HTML Tags and Attributes**

If you dynamically create HTML tags and construct tag attributes with potentially unsafe input, make sure you HTML-encode the tag attributes before writing them out.

The following .aspx page shows how you can write HTML directly to the return page by using the <**asp:Literal**> control. The code takes user input of a color name, inserts it into the HTML sent back, and displays text in the color entered. The page uses **HtmlEncode** to ensure the inserted text is safe.

<%@ Page Language="C#" AutoEventWireup="true"%>

<html>

<form id="form1" runat="server">

<div>

Color:&nbsp;<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox><br />

<asp:Button ID="Button1" runat="server" Text="Show color"

OnClick="Button1\_Click" /><br />

<asp:Literal ID="Literal1" runat="server"></asp:Literal>

</div>

</form>

</html>

<script runat="server">

private void Page\_Load(Object Src, EventArgs e)

{

protected void Button1\_Click(object sender, EventArgs e)

{

Literal1.Text = @"<span style=""color:"

+ Server.HtmlEncode(TextBox1.Text)

+ @""">Color example</span>";

}

}

</Script>

**Potentially Dangerous HTML Tags**

While not an exhaustive list, the following commonly used HTML tags could allow a malicious user to inject script code:

* <**applet**>
* <**body**>
* <**embed**>
* <**frame**>
* <**script**>
* <**frameset**>
* <**html**>
* <**iframe**>
* <**img**>
* <**style**>
* <**layer**>
* <**link**>
* <**ilayer**>
* <**meta**>
* <**object**>

An attacker can use HTML attributes such as **src**, **lowsrc**, **style**, and **href** in conjunction with the preceding tags to inject cross-site scripting. For example, the **src** attribute of the <**img**> tag can be a source of injection, as shown in the following examples.

<img src="javascript:alert('hello');">

<img src="java&#010;script:alert('hello');">

<img src="java&#X0A;script:alert('hello');">

An attacker can also use the <**style**> tag to inject a script by changing the MIME type as shown in the following.

<style TYPE="text/javascript">

alert('hello');

</style>

**Step 5. Evaluate Countermeasures**

When you find ASP.NET code that generates HTML using some input, you need to evaluate appropriate countermeasures for your specific application. Countermeasures include:

* **Encode HTML output.**
* **Encode URL output.**
* **Filter user input.**

**Encode HTML Output**

If you write text output to a Web page and you do not know if the text contains HTML special characters (such as <, >, and **&**), pre-process the text by using the **HttpUtility.HtmlEncode** method as shown in the following code example. Do this if the text came from user input, a database, or a local file.

Response.Write(HttpUtility.HtmlEncode(Request.Form["name"]));

Do not substitute encoding output for checking that input is well-formed and correct. Use it as an additional security precaution.

**Encode URL Output**

If you return URL strings that contain input to the client, use the **HttpUtility.UrlEncode** method to encode these URL strings as shown in the following code example.

Response.Write(HttpUtility.UrlEncode(urlString));

**Filter User Input**

If you have pages that need to accept a range of HTML elements, for example through some kind of rich text input field, you must disable ASP.NET request validation for the page. If you have several pages that do this, create a filter that allows only the HTML elements that you want to accept. A common practice is to restrict formatting to safe HTML elements such as bold (<**b**>) and italic (<**i**>).

**To safely allow restricted HTML input**

1. Disable ASP.NET request validation by the adding the **ValidateRequest="false"** attribute to the **@ Page** directive.
2. Encode the string input with the **HtmlEncode** method.
3. Use a **StringBuilder** and call its **Replace** method to selectively remove the encoding on the HTML elements that you want to permit.

The following .aspx page code shows this approach. The page disables ASP.NET request validation by setting **ValidateRequest="false"**. It HTML-encodes the input and then selectively allows the <**b**> and <**i**> HTML elements to support simple text formatting.

<%@ Page Language="C#" ValidateRequest="false"%>

<script runat="server">

void submitBtn\_Click(object sender, EventArgs e)

{

// Encode the string input

StringBuilder sb = new StringBuilder(

HttpUtility.HtmlEncode(htmlInputTxt.Text));

// Selectively allow  **<b> and <i>**

**sb.Replace("&lt;b&gt;", "<b>");**

**sb.Replace("&lt;/b&gt;", "**");

sb.Replace("&lt;i&gt;", "<i>");

sb.Replace("&lt;/i&gt;", "");

Response.Write(sb.ToString());

}

</script>

<html>

<body>

<form id="form1" runat="server">

<div>

<asp:TextBox ID="htmlInputTxt" Runat="server"

TextMode="MultiLine" Width="318px"

Height="168px"></asp:TextBox>

<asp:Button ID="submitBtn" Runat="server"

Text="Submit" OnClick="submitBtn\_Click" />

</div>

</form>

</body>

</html>

**Additional Considerations**

In addition to the techniques discussed previously in this How To, use the following countermeasures as further safe guards to prevent cross-site scripting:

* **Set the correct character encoding**.
* **Do not rely on input sanitization**.
* **Use the HttpOnly cookie option**.
* **Use the** <**frame**> **security attribute**.
* **Use the innerText property instead of innerHTML**.

**Set the Correct Character Encoding**

To successfully restrict valid data for your Web pages, you should limit the ways in which the input data can be represented. This prevents malicious users from using canonicalization and multi-byte escape sequences to trick your input validation routines. A multi-byte escape sequence attack is a subtle manipulation that uses the fact that character encodings, such as uniform translation format-8 (UTF-8), use multi-byte sequences to represent non-ASCII characters. Some byte sequences are not legitimate UTF-8, but they may be accepted by some UTF-8 decoders, thus providing an exploitable security hole.

ASP.NET allows you to specify the character set at the page level or at the application level by using the <**globalization**> element in the Web.config file. The following code examples show both approaches and use the ISO-8859-1 character encoding, which is the default in early versions of HTML and HTTP.

To set the character encoding at the page level, use the <**meta**> element or the **ResponseEncoding** page-level attribute as follows:

<meta http-equiv="Content Type"

content="text/html; charset=ISO-8859-1" />

OR

<% @ Page ResponseEncoding="iso-8859-1" %>

To set the character encoding in the Web.config file, use the following configuration.

<configuration>

<system.web>

<globalization

requestEncoding="iso-8859-1"

responseEncoding="iso-8859-1"/>

</system.web>

</configuration>

**Validating Unicode Characters**

Use the following code to validate Unicode characters in a page.

using System.Text.RegularExpressions;

. . .

public class WebForm1 : System.Web.UI.Page

{

private void Page\_Load(object sender, System.EventArgs e)

{

// Name must contain between 1 and 40 alphanumeric characters

// and (optionally) special characters such as apostrophes

// for names such as O'Dell

if (!Regex.IsMatch(Request.Form["name"],

@"^[\p{L}\p{Zs}\p{Lu}\p{Ll}\']{1,40}$"))

throw new ArgumentException("Invalid name parameter");

// Use individual regular expressions to validate other parameters

. . .

}

}

The following explains the regular expression shown in the preceding code:

* **^** means start looking at this position.
* **\p{ ..}** matches any character in the named character class specified by **{..}**.
* **{L}** performs a left-to-right match.
* **{Lu}** performs a match of uppercase.
* **{Ll}** performs a match of lowercase.
* **{Zs}** matches separator and space.
* **'**matches apostrophe.
* **{1,40}** specifies the number of characters: no less than 1 and no more than 40.
* $ means stop looking at this position.

**Do Not Rely on Input Sanitization**

A common practice is for code to attempt to sanitize input by filtering out known unsafe characters. Do not rely on this approach because malicious users can usually find an alternative means of bypassing your validation. Instead, your code should check for known secure, safe input. Table 1 shows various safe ways to represent some common characters.

**Table 1: Character Representation**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Characters | Decimal | Hexadecimal | HTML Character Set | Unicode |
| **"** (double quotation marks) | &#34 | &#x22 | &quot; | \u0022 |
| **'** (single quotation mark) | &#39 | &#x27 | &apos; | \u0027 |
| **&** (ampersand) | &#38 | &#x26 | &amp; | \u0026 |
| < (less than) | &#60 | &#x3C | &lt; | \u003c |
| > (greater than) | &#62 | &#x3E | &gt; | \u003e |

**Use the HttpOnly Cookie Option**

Internet Explorer 6 Service Pack 1 and later supports an **HttpOnly** cookie attribute, which prevents client-side scripts from accessing a cookie from the **document.cookie** property. Instead, the script returns an empty string. The cookie is still sent to the server whenever the user browses to a Web site in the current domain.

**Note** Web browsers that do not support the **HttpOnly** cookie attribute either ignore the cookie or ignore the attribute, which means that it is still subject to cross-site scripting attacks.

The **System.Net.Cookie** class in Microsoft .NET Framework version 2.0 supports an **HttpOnly** property. The **HttpOnly** property is always set to true by Forms authentication.

Earlier versions of the .NET Framework (versions 1.0 and 1.1) require that you add code similar to the following to the **Application**\_**EndRequest** event handler in your application Global.asax file to explicitly set the **HttpOnly** attribute.

protected void Application\_EndRequest(Object sender, EventArgs e)

{

string authCookie = FormsAuthentication.FormsCookieName;

foreach (string sCookie in Response.Cookies)

{

// Just set the HttpOnly attribute on the Forms

// authentication cookie. Skip this check to set the attribute

// on all cookies in the collection

if (sCookie.Equals(authCookie))

{

// Force HttpOnly to be added to the cookie header

Response.Cookies[sCookie].Path += ";HttpOnly";

}

}

}

**Use the <frame> Security Attribute**

Internet Explorer 6 and later support a new **security** attribute for the <**frame**> and <**iframe**> elements. You can use the **security** attribute to apply the user's Restricted Sites Internet Explorer security zone settings to an individual frame or iframe. By default, the Restricted Sites zone does not support script execution.

If you use the **security** attribute, it must be set to **"restricted"** as shown in the following.

<frame security="restricted" src="http://www.somesite.com/somepage.htm"></frame>

**Use the innerText Property Instead of innerHTML**

If you use the **innerHTML** property to build a page and the HTML is based on potentially untrusted input, you must use **HtmlEncode** to make it safe. To avoid having to remember to do this, use **innerText** instead. The **innerText** property renders content safe and ensures that scripts are not executed.

The following example shows this approach for two HTML <**span**> controls. The code in the **Page\_Load** method sets the text displayed in the **Welcome1** <**span**> element using the **innerText** property, so HTML-encoding is unnecessary. The code sets the text in the **Welcome2** <**span**> element by using the **innerHtml** property; therefore, you must **HtmlEncode** it first to make it safe.

<%@ Page Language="C#" AutoEventWireup="true"%>

<html>

<body>

<span id="Welcome1" runat="server"> </span>

<span id="Welcome2" runat="server"> </span>

</body>

</html>

<script runat="server">

private void Page\_Load(Object Src, EventArgs e)

{

// Using InnerText renders the content safe–no need to HtmlEncode

Welcome1.InnerText = "Hello, " + User.Identity.Name;

// Using InnerHtml requires the use of HtmlEncode to make it safe

Welcome2.InnerHtml = "Hello, " +

Server.HtmlEncode(User.Identity.Name);

}

</Script>