# Evaluating and Reducing the Exposure to Type 1 cross-site scripting (XSS) attacks using secure development practices

**Syed Nisar Hussain Bukhari [1] (Scientist-B), Ashaq Hussain Dar [2] (Scientist-C)**

National Institute of Electronics and Information Technology, J & K,

Department of Electronics and IT, Ministry of communication and Information Technology, Govt. of India

E-mail: nisar.bukhari@gmail.com [1], ashaqhussain@yahoo.com [2]

## Abstract

As the use of the Internet has grown, so has the number of attacks which attempt to use it for nefarious purposes. One vulnerability which has become commonly exploited is known as cross-site scripting (XSS). An attack on this class of vulnerabilities occurs when an attacker injects malicious code into a web application in an attempt to gain access to unauthorized information. In such instances, the victim is unaware that their information is being transferred from a site that he/she trusts to another site controlled by the attacker. In this paper we shall focus on type 1 or "non-persistent cross-site scripting". With non-persistent cross-site scripting, malicious code or script is embedded in a Web request, and then partially or entirely echoed (or "reflected") by the Web server without encoding or validation in the Web response. The malicious code or script is then executed in the client's Web browser which could lead to several negative outcomes, such as the theft of session data and accessing sensitive data within cookies. In order for this type of cross-site scripting to be successful, a malicious user must coerce a user into clicking a link that triggers the non-persistent cross-site scripting attack. This is usually done through an email that encourages the user to click on a provided malicious link, or to visit a web site that is fraught with malicious links. In this paper type 1 or "non-persistent cross-site scripting" attack shall be evaluated. We will also show how these attacks can be reduced using secure development practices.

Keywords: cross-site scripting, XSS, non-persistent, attack.

## 1. Introduction

JavaScript is a powerful tool for developing rich Web applications. Without client-side execution of code embedded in HTML and XHTML pages, the dynamic nature of Web applications like Google Maps, Try Ruby! and Zoho Office and so would not be possible. Unfortunately, any time you add complexity to a system, you increase the potential for security issues -- and adding JavaScript to a Web page is no exception. Among the problems introduced by JavaScript are:

1. A malicious website might employ JavaScript to make changes to the local system, such as copying or deleting files.
2. A malicious website might employ JavaScript to monitor activity on the

local system, such as with keystroke logging.

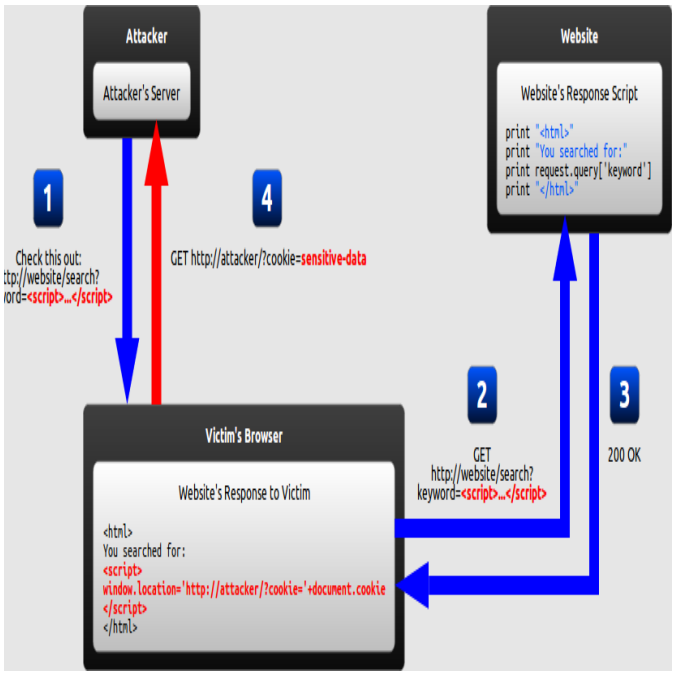
3. A malicious website might employ JavaScript to interact with other Websites the user has open in other browser windows or tabs.

The first and second problems in the above list can be mitigated by turning the browser into a sort of "sandbox" that limits the way JavaScript is allowed to behave so that it only works within the browser's little world. The third can be limited somewhat as well, but it is all too easy to get around that limitation because whether a particular webpage can interact with another webpage in a given manner may not be something that can be controlled by the software employed by the end user. Sometimes, the ability of one website's JavaScript to steal data meant for another Website can only be limited by the due diligence of the other website's developers.

The key to defining cross-site scripting is in the fact that vulnerabilities in a given website's use of dynamic Web design elements may give someone the opportunity to use JavaScript for security compromises. It's called "cross-site" because it involves interactions between two separate websites to achieve its goals. In many cases, however, even though the exploit involves the use of JavaScript, the website that's vulnerable to cross-site scripting exploits does not have to employ JavaScript itself at all. Only in the case of local cross-site scripting exploits does the vulnerability have to exist in JavaScript sent to the browser by a legitimate website. [1]

## 2. Type 1 or Non persistent XSS attack scenario

In a reflected/type 1 XSS attack, the malicious string is part of the victim's request to the website. The website then includes this malicious string in the response sent back to the user. The diagram below illustrates this scenario:




1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website includes the malicious string from the URL in the response.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.

### 3. Reflected XSS attack seems harmless-But it isn't!!

At first, reflected XSS might seem harmless because it requires the victim himself to actually send a request containing a malicious string. Since nobody would willingly attack himself, there seems to be no way of actually performing the attack. As it turns out, there are at least two common ways of causing a victim to launch a reflected XSS attack against himself:

- Attack occurs when an attacker takes advantage of such applications and creates a request with malicious data (such as a script) that is later presented to the user requesting it. The malicious content is usually embedded into a hyperlink, positioned so that the user will come across it in a web site, a Web message board, an email, or an instant message.[5]
- If the user targets a large group of people, the attacker can publish a link to the malicious URL (on his own website or on a social network, for example) and wait for visitors to click it.

These two methods are similar, and both can be more successful with the use of a URL shortening, which masks the malicious string from users who might otherwise identify it.[2]

### 4. Statistics showing the frequency of Type 1 XSS attack

Sites continue to fall prey to XSS attacks because most need to be interactive, accepting and returning data from users [3]. As per the report by Search Security organization team the penetration testing work conducted at Intelguardians, approximately 80% of the Web applications they tested have XSS flaws [6]. It has been around since the 1990s and most major websites like Google, Yahoo and Facebook have all been affected by cross-site scripting flaws at some point. According to a latest WhiteHat Security Statistics Report 86% of all websites had at least one serious vulnerability but cross-site scripting is the most frequently found serious vulnerability. Of the total population of the vulnerabilities identified, Cross-Site Scripting, Information Leakage and Content Spoofing took the top three spots at 43%, 11% and 13% respectively. This is near linear repeat of 2011 where the percentage was 50%, 14% and 9% [4]

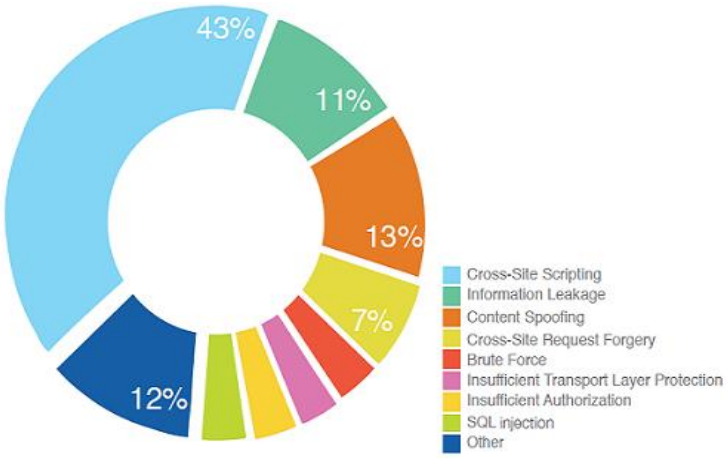



Fig 2: Overall vulnerability population (2012).%age breakdown of all serious vulnerabilities discovered

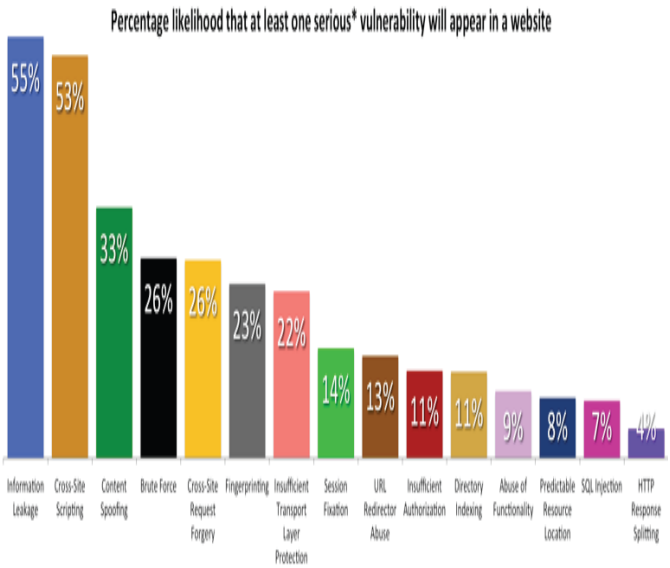



Fig 3: Top 15 vulnerability classes (2012) -sorted by vulnerability class.

## 5. Research Findings

Most of the web developers particularly those having less experience or no experience at all write in secure code. I reviewed as part of my work the source code of around ten web applications and I found that developers are not aware that their applications are open to simple script injection attacks. Whether the purpose of these attacks is to deface the site by displaying HTML, or to potentially execute client script to redirect the user to a hacker's site, script injection attacks are a problem that Web developers must contend with. Script injection attacks are a concern of all web developers, whether they are using ASP.NET, ASP, or other web development technologies. The great thing about the ASP.NET web technology is that request validation feature proactively prevents these attacks by not allowing unencoded HTML content to be processed by the server unless the developer decides to allow that content.

I asked the developers for testing and I have found that if they enter something which includes html tags or like tokens as <h1> hello</h1> in an input field such as text box, they get the HttpRequestValidationException because of the ValidateRequest option which is a part of the built-in protection mechanism with ASP.NET. This feature can be enabled on a per-page basis, or globally through web.config file settings. This option, when set to "true," instructs ASP.NET to inspect all inputs into a Web-based application for potentially dangerous inputs. If any potentially dangerous inputs are detected, then HttpRequestValidationException is thrown and the attack is halted. This may be an attempt to compromise the security of your application, such as a cross-site scripting attack. I have experienced that based on the message received after such an attempt they immediately override application request validation settings by setting the requestValidationMode attribute in the httpRuntime configuration section to requestValidationMode="2.0".After setting this value, disabling request validation by setting validateRequest="false" in the Page directive or in the <pages> configuration section the request is easily processed which is security threat. So to mention here, it is strongly recommended that your application explicitly check all inputs in this case.

**Caution**: when request validation is disabled, content can be submitted to your application; it is the responsibility of the application developer to ensure that content is properly encoded or processed. So to

reduce the risk from cross-site scripting attacks, developers need to transform or neutralize user input that may contain potentially executable code or script into non-executable forms. That is, the Web browser needs to be told in some way that the following data is not executable code and should treated as data only. The way this transformation or neutralization is achieved is through encoding. Encoding will automatically replace any '<' or '>' (together with several other symbols) with their corresponding HTML encoded representation. For example, '<' is replaced by '<' and '>' is replaced by '>' Browsers use these special codes to display the '<' or '>' in the browser. Out of ten applications I found only two applications using encoding as Server.HtmlEncode API and writing secure code which clearly states that around 80% of web applications do have XSS and other types of vulnerabilities. This closely matches with the report prepared by Search Security organization team and WhiteHat Security.

## 6. Reducing the Exposure using secure development practices

There are several measures you can take as a developer to reduce the exposure to cross-site scripting attacks conducted through your Web-based applications.

- The first defensive measure which can be applied to address a majority of application security vulnerabilities is input validation. Ensure that all untrusted inputs into Web-based applications conform to the expected input formats.
- Check for correctness with format, length, type, and range. Example sources of untrusted input include, but are not limited to, data from users, data from a database, or data from an un-trusted Web service.
- Encode any Web response data that may contain user input or other untrusted input
- Web-based applications built using Microsoft ASP.NET can leverage built-in protection via the ValidateRequest option.
- Another defensive measure that can be used to help protect applications from cross-site scripting attacks is the Microsoft Anti-Cross Site Scripting Library (AntiXSS). This library provides additional encoding capabilities not provided by the standard encoding libraries included in the .NET Framework [7].
- The .NET Framework has built-in encoding libraries under the class System.Web.HttpUtility. The encoding methods in this class work by looking for specific characters that are common in cross-site scripting attacks and encode them into non-executable forms [7].
- The Microsoft Anti-Cross Site Scripting Library takes a different approach by first defining a set of valid characters, and then encoding any characters not in that valid set. Both are effective in reducing exposure to a majority of cross-site scripting attacks; however, they differ in the method in which they reduce exposure.

## 7. Conclusion

Cross-site scripting vulnerabilities are the most frequently encountered Web-based vulnerabilities today, and have been found on several major Web sites. These vulnerabilities manifest in Web-based applications whenever best practices, such as input validation, and Web output encoding are not implemented in code. To reduce exposure to these attacks, developers should implement a multi-layer defense strategy that includes coding best practices such as input validation, Web output encoding, and leveraging built-in platform protection. Microsoft has better enabled developers to do so through the guidance, process and tools of the Microsoft SDL.

## References


[1] www.techrepublic.com

[2] http://excess-xss.com/

[3] http://www.computerweekly.com

[4] White Hat Security, "Website statistics report"

[5] Cross-site scripting attack prevention, www.imperva.com

[6] What new tactics can prevent XSS attacks, www.searchsecurity.techtarget.com

[7] Microsoft SDL-Developer starter Kit, (Cross site scripting level-200), www.microsoft.com/sdl