EXTENUATING CROSS-SITE SCRIPTING ATTACKS

V.Shanmughaneethi
Lecturer, Computer Centre
National Institute of Technical Teachers Training and Research
Taramani, Chennai 113

Dr. S.Sawmynathan,
Asst. Professor, Dept., of CSE,
Anna University,
Chennai 25

Abstract

Websites rely heavily on complex web applications to deliver different output or content to a wide variety of users according to set preferences and specific needs. This arms organizations with the ability to provide better value to their customers and prospects. Today, Most of the Websites suffer from serious vulnerabilities like Buffer Overflow, Information Leakage, SQL Injection, Denial of Service and Cross-site Scripting. So that, the rendering organizations helpless to provide the secure information to their clients. Web applications are designed to allow any user with a web browser and an internet connection to interact with them in a platform independent way. They are typically constructed in a two- or three-tiered architecture consisting of at least an application running on a web server, and a back-end database. Frequently most of the applications make use of JavaScript code that is embedded into web pages to support dynamic client-side behavior.

This script code is executed in the context of the user's web browser. To protect the user's environment from malicious JavaScript code, a sandboxing mechanism is used that limits a program to access only resources associated with its origin site. Unfortunately, these security mechanisms fail if a user can be lured into downloading malicious JavaScript code from an intermediate, trusted site. In this case, the malicious script is granted full access to all resources (e.g., authentication tokens and cookies) that belong to the trusted site. So, preventing the attacks is a major problem in web application scenario. By implementing proper security measure, web applications are secured as well as by providing a secure environment in which users are comfortable working with the web applications. Hence, today web application security is finally getting more prominent attention.

Scope

The World Wide Web is one of the most exciting and useful applications of the Internet. But, as is often the case, the designers of the WWW did not adequately consider protection and security when implementing the service; they opted for complete openness. As a result, the demand for security services for potential users has grown rapidly. Modern applications such as electronic commerce, business transactions, and information sharing have driven towards the development of many different approaches to provide security

capabilities on the WWW. Hackers are constantly experimenting with a wide repertoire of hacking techniques to compromise websites and web applications and make off with a treasure trove of sensitive data including credit card numbers, social security numbers and even medical records. Common Threat in the web applications are Abuse of Functionality, Brute Force, Buffer Overflow, Content Spoofing, Credential / Session Prediction, Cross-site Scripting, Denial of Service, Directory Indexing, Format String Attack, Information Leakage, Insufficient Anti-automation, Insufficient Authentication, Insufficient Authorization, Insufficient Process Validation, Insufficient Session Expiration, LDAP Injection, OS Commanding, Path Traversal, Predictable Resource Location, Session Fixation, SQL Injection, SSI Injection, Weak Password Recovery Validation, and XPath Injection.

Among the variety of web attacks Cross Site Scripting (also known as XSS or CSS) is generally believed to be one of the most common application layer hacking techniques.

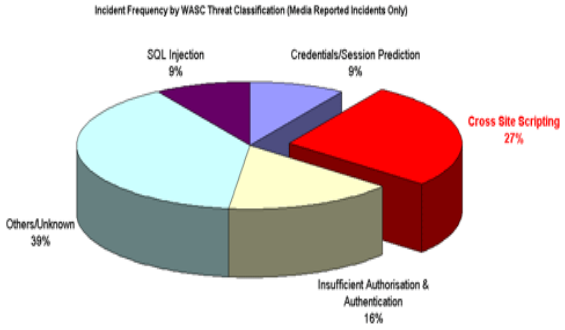

Fig .1 [7]

Web pages contain both text and HTML markup that is generated by the server and interpreted by the client browser. Servers that generate static pages have full control over how the client will interpret the pages sent by the server. However, servers that generate dynamic pages do not have complete control over how their output is interpreted by the client. The heart of the issue is that if un-trusted content can be introduced into a dynamic page, neither the server nor the client has enough information to recognize that this has

happened and take protective actions. Cross-site scripting refers to that hacking technique that leverages vulnerabilities in the code of a web application. This allows an attacker to embed malicious JavaScript, VBScript, ActiveX, HTML, or Flash into a vulnerable dynamic page to fool the user, executing the script on his machine in order to gather data. The use of XSS might compromise private information, manipulate or steal cookies, create requests that can be mistaken for those of a valid user, or execute malicious code on the end-user systems. The data is usually formatted as a hyperlink containing malicious content and which is distributed over any possible means on the internet.

As a hacking tool, the attacker can formulate and distribute a custom-crafted CSS URL just by using a browser to test the dynamic website response. The attacker also needs to know some HTML, JavaScript and a dynamic language, to produce a URL which is not too suspicious-looking, in order to attack a XSS vulnerable website. In general, XSS attacks are easy to execute, but difficult to detect and prevent. One reason is the high flexibility of HTML encoding schemes, offering the attacker many possibilities for circumventing server-side input filters that should prevent malicious scripts from being injected into trusted sites. Also, devising a client-side solution is not easy because of the difficulty of identifying JavaScript code as being malicious.

Example of a Cross Site Scripting attack

As a simple example, imagine a search engine site which is open to an XSS attack. The query screen of the search engine is a simple single field form with a submit button. Whereas the results page, displays both the matched results and the text you are looking for.

Example:

Search Results for "XSS Vulnerability"[7]

To be able to bookmark pages, search engines generally leave the entered variables in the URL address. In this case the URL would look like:

http://test.searchengine.com/search.php?q=XSS%20Vulnerability

Next try to send the following query to the search engine:

<script type="text/javascript"> alert('This is an XSS Vulnerability') </script>

By submitting the query to search.php, it is encoded and the resulting URL would be something like:

http://test.searchengine.com/search.php?q=%3Cscript%3Ealert%28%91This%20is%20an%20XSS%20Vulnerability%92%29%3C%2Fscript%3E

Upon loading the results page, the test search engine would probably display no results for the search but it will display a JavaScript alert which was injected into the page by using the XSS vulnerability.

Types of XSS Attacks [2]

Two main classes of XSS attacks exist: *stored* attacks and *reflected* attacks. In a stored XSS attack, the malicious JavaScript code is permanently stored on the target server (e.g., in a database, in a message forum, in a guestbook, etc.). In a reflected XSS attack, on the other hand, the injected code is "reflected" off the web server such as in an error message or a search result that may include some or all of the input sent to the server as part of the request. Reflected XSS attacks are delivered to the victims via e-mail messages or links embedded on other web pages. When a user clicks on a malicious link or submits a specially crafted form, the injected code travels to the vulnerable web application and is reflected back to the victim's browser. There are a number of input validation and filtering techniques that web developers can use in order to prevent XSS vulnerabilities. However, these are *server-side solutions over which the end user has no control*. The easiest and the most effective client-side solution to the XSS problem for users is to deactivate JavaScript in their browsers. Unfortunately, this solution is often not feasible because a large number of web sites use JavaScript for navigation and enhanced presentation of information. Thus, a novel solution to the XSS problem is necessary to allow users to execute JavaScript code in a more secure fashion.

Check Cross Site Scripting Vulnerabilities

Web application is vulnerable to cross-site scripting attacks wherever it uses input parameters in the output HTML stream returned to the client. Even before to conduct a code review, run a simple test to check if the application is vulnerable to XSS. Search for pages where user input information is sent back to the browser.

XSS bugs are an example of maintaining too much trust in data entered by a user. For example, your application might expect the user to enter a price, but instead the attacker includes a price and some HTML and JavaScript. Therefore, always ensure that data that comes from un-trusted sources is validated. When reviewing code, always ask the question, "Is this data validated?" Keep a list of all entry points into the application, such as HTTP headers, query strings, form data, and so on, and make sure that all input is checked for validity at some point. The most common way to check that data is valid is to use regular expressions. To 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, then your Web application is vulnerable to XSS.

Methods to Identify and Protect Common XSS [1]

1. Identify Code that Outputs Input
2. Searching for ".Write"
3. Identify Potentially Dangerous HTML Tags and Attributes and Special Characters
4. Identify Code That Handles URLs
5. Encoding dynamic output elements
6. Explicitly setting the character set encoding for each page generated by the web server
7. Check the HttpOnly Cookie Option
8. Check the <frame> Security Attribute
9. Check the Use of the innerText and innerHTML Properties
10. Filtering specific characters in dynamic elements
11. Examine cookies

Identify Code that Outputs Input

View the page output source from the browser to see if the code is placed inside an attribute. If it is, inject the following code and retest to view the output. "onmouseover= alert('hello');" A common technique used by developers is to filter for < and > characters. If the code that you review filters for these characters, then test using the following code instead: &{alert('hello');} If the code does not filter for those characters, then you can test the code by using the following script: <script> alert(document.cookie); </script>; You may have to close a tag before using this script, "> <script> alert(document.cookie);</script>

Searching for ".Write"

Search for the ".Write" string across web application source code and code contained in any additional assembly you have developed for your application. This locates occurrences of Response. Write, and any internal routines that may generate output through a response object variable,

```
public void WriteOutput(Response respObj)
{  respObj.Write(Request.Form["someField"]); }
```

You should also search for the "<%=" string within web application source code, which can also be used to write output, <%=myVariable %>

Identify Potentially Dangerous HTML Tags, Attributes and Special Characters

While not exhaustive, 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> <ilayer> <meta> <object>

HTML attributes such as src, lowsrc, style, and href can be used in conjunction with the tags above to cause XSS. For example, the src attribute of the <img> tag can be a source of injection as like

```
<IMG SRC="javascript:alert('hello');">
<IMG SRC="javascript:alert('hello');">
<IMG SRC="javascript:alert('hello');">
```

The <style> tag also can be a source of injection by changing the MIME type is

<style TYPE="text/javascript"> alert('hello'); </style>

For encoding and filtering, requires an understanding of "special characters". The HTML specification determines which characters are "special", because they have an effect on how the page is displayed. However, many web browsers try to correct common errors in HTML. As a result, they sometimes treat characters as special when, according to the specification, they aren't. In addition, the set of special characters depends on the context:

- In the content of a block-level element (in the middle of a paragraph of text) "<" is special because it introduces a tag. "&" is special because it introduces a character entity. ">" is special because some browsers treat it as special, on the assumption that the author of the page really meant to put in an opening "<", but omitted it in error. In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL. When this is done, it introduces additional special characters:

Space, tab, and new line are special because they mark the end of the URL.
"&" is special because it introduces a character entity or separates CGI parameters.
Non-ASCII characters (that is, everything above 128 in the ISO-8859-1 encoding) aren't allowed in URLs, so they are all special here.
The "%" must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server-side code. The percent must be filtered if input such as %68%65%6C%6C%6F" becomes "hello" when it appears on the web page in question.

- Within the body of a <SCRIPT> </SCRIPT>

The semicolon, parenthesis, curly braces, and new line should be filtered in situations where text could be inserted directly into a preexisting script tag.

- Server-side scripts

Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering.

- Other possibilities

No current exploits rely on the ampersand. This character may be useful in future exploits. Conservative web page authors should filter this character out if possible.

It is important to note that individual situations may warrant including additional characters in the list of special characters. Web developers must examine their applications and determine which characters can affect their web applications.

Identify Code that Handles URLs

Code that handles URLs can be vulnerable. Review the code to see if it is vulnerable to the following common attacks: If a Web server is not up-to-date with the latest security patches, it could be vulnerable to directory traversal and double slash attacks, such as: http://www.myweb.com/..%255%../winnt. http://www.myweb.com/..%255%..//somedirectory.

If your code filters for "/", an attacker can easily bypass the filter by using an alternate representation for the same character. For example, the overlong UTF-8 representation of "/" is "%c0f%af" and this could be used in the following URL: http://www.myweb/..%c0f%af../winnt. If the code processes query string input, check that it constrains the input data and performs bounds checks. Check that the code is not vulnerable if an attacker passes an extremely large amount of data through a query string parameter. http://www.myweb/test.aspx?var=InjectHugeAmountOfDataHere

Encoding Dynamic Output Elements

Each character in the ISO-8859-1 specification can be encoded using its numeric entry value. The following example uses the copyright mark in an HTML document: <p>&#169 2008 Some Co., Inc. The copyright character is 169 and using the &# syntax allows the author to insert encoded characters that will be interpreted by the browser. In addition, many of the ISO-8859-1 characters include an entity name encoding. The copyright can also be done using this method: <p>© 2008 Some Co., Inc. Encoding untrusted data has benefits over filtering untrusted data, including the preservation of visual appearance in the browser. This is important when special characters are considered acceptable. Unfortunately, encoding all untrusted data can be resource intensive. Web developers must select a balance between encoding and the other option of data filtering.

Explicitly Setting the Character Encoding

Many web pages leave the character encoding undefined. In earlier versions of HTML and HTTP, the character encoding was supposed to default to ISO-8859-1 if it wasn't defined. In fact, many browsers had a different default, so it was not possible to rely on the default being ISO-8859-1.

If the web server doesn't specify which character encoding is in use, it can't tell which characters are special. Web pages with unspecified character encoding work most of the time because most character sets assign the same characters to byte values below 128. But which of the values above 128 are special? Some 16-bit character-encoding schemes have additional multi-byte representations for special characters such as "<". Some browsers recognize this alternative encoding and act on it. This is "correct" behavior, but it makes attacks using malicious scripts much harder to prevent. The server simply doesn't know which byte sequences represent the special characters.

For example, UTF-7 provides alternative encoding for "<" and ">", and several popular browsers recognize these as the start and end of a tag. This is not a bug in those browsers. If the character encoding really is UTF-7, then this is correct behavior. The problem is that it is possible to get into a situation in which the browser and the server disagree on the encoding. Web servers should set the character set, then make sure that the data they insert is free from byte sequences that are special in the specified encoding.

Example:

<HTML><HEAD><META http-equiv="Content-Type"content="text/html; charset=ISO-8859-1">
<TITLE>HTML SAMPLE</TITLE></HEAD>
<BODY><P>This is a sample HTML page</BODY>
</HTML>

The META tag in the HEAD section of this sample HTML forces the page to use the ISO-8859-1 character set encoding.

Check the HttpOnly Cookie Option

Internet Explorer 6 SP 1 supports a new HttpOnly cookie attribute that prevents client-side script from accessing the cookie from the document.cookie property. Instead, an empty string is returned. The cookie is still sent to the server whenever the user browses to a Web site in the current domain.

Check the <frame> Security Attribute

Internet Explorer 6 and later supports a new security attribute on the <frame> and <iframe> elements. Developer can use the security attribute to apply the user's Restricted Sites Internet Explorer security zone settings to an individual frame or iframe.

Check the Use of the innerText and innerHTML Properties

If you create a page with untrusted input, verify that you use the innerText property instead of innerHTML. The innerText property renders content safe and ensures that script is not executed.

Filtering Dynamic Content

Unfortunately, it is unclear whether there are any other characters or character combinations that can be used to expose other vulnerabilities. The recommended method is to select the set of characters that is known to be safe rather than excluding the set of characters that might be bad. For example, a form element that is expecting a person's age can be limited to the set of digits 0 through 9. There is no reason for this age element to accept any letters or other special characters. Using this positive approach of selecting the characters that are acceptable will help to reduce the ability to exploit other yet unknown vulnerabilities.

The filtering process can be done as part of the data input process, the data output process, or both. Filtering the data during the output process, just before it is rendered as part of the dynamic page, is recommended. Done correctly, this approach ensures that all dynamic content is filtered. Filtering on the input side is less effective because dynamic content can be entered into a web sites database(s) via methods other than HTTP. In this case, the web server may never see the data as part of the input process. Unless the filtering is implemented in all places where dynamic data is entered, the data elements may still be remain tainted.

Examine Cookies

One method to exploit this vulnerability involves inserting malicious content into a cookie. Web developers should carefully examine cookies that they accept and use the filtering techniques describe above to verify that they are not storing malicious content.

Conclusion

XSS vulnerabilities are being discovered and disclosed at an alarming rate. XSS attacks are generally simple, but difficult to prevent because of the high flexibility that HTML encoding schemes provide to the attacker for circumventing server-side input filters. In this paper, we have presented the formal approaches to protect cross site scripting attacks in web applications.

References

1. http://www.cert.org/
2. http://www.securityfocus.com
3. http://www.webappsec.org/articles/
4. http://www.webappsec.org/projects/articles/guidelines.shtml
5. www.webappsec.org
6. Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web pplication code by static analysis and runtime protection. In World WideWeb, pages
7. http://www.acunetix.com/websitesecurity/xss.html