Global Information Assurance Certification Paper

Preventing Cross-Site Scripting Vulnerability

Deyu Hu
May 4th, 2004

GSEC Version 1.4b (Option 1)

Abstract

The cross-site scripting attack method was first discussed in a CERT advisory[1]. Since then, many examples[2] [3] of the attack were published and prevention and mitigation methods[4] [5] [6] discussed. It is becoming clear that this type of attack has become one of the top security flaws within existing web applications[7].

This paper focuses on prevention techniques, especially in J2EE based development environments involving Java/JSP/Servlet technologies. It explains in detail how the published methods can be applied in such environments. In addition, it discusses situations where developers need to accept certain "safe" markup content from users, while rejecting "unsafe" inputs.

Introduction

Cross-site scripting is an attack against dynamic web sites that interact with end users via user inputs (CSS or XSS has been used as an abbreviation for this attack). HTML pages generated by these web sites generally contain a mixture of static information generated by the web sites and dynamic information originated from end users. A common secure programming principal says that any user input must be validated. In this case, if user inputs are directly placed into the HTML pages without validation, it is possible for malicious users to inject dynamic programs in the form of JavaScript or VBScript into these pages.

Given that the attack mechanisms are well understood today, it should be pointed out that, when possible, a proper strategy should be developed to counter such an attack from the very beginning of the software development cycle. It is often much more costly to test and fix the problem afterwards. Although testing and fixing a specific instance of the problem is relatively simple and straightforward, it is quite difficult to be sure that a complex web site consisting of tens of thousands of dynamic pages is immune to this attack. Automated testing tools for this type of problem has not yet matured.

A side benefit for a web site to systematically prevent this attack is that the site becomes more robust even in the absence of malicious users. Dynamically generated pages can fail for other reasons when input is not properly validated. For example, if a user input contains any markup character (such as '<', '>', etc), without proper treatment, these characters may interfere with the HTML the server generates, resulting in broken user interfaces.

The rest of the paper is organized as follows: the next section explains the detailed cross-site scripting attack mechanism. It is followed by explanation of three common prevention methods. The last section concludes.

Cross-site scripting attack mechanism

Users interact with a dynamic web site by clicking on links or filling in and submitting forms, which results in a list of name/value pairs being sent to the server in the form of an http request. The request can contain other information such as a list of cookies, the referrer URL, etc. In general, any data in the request should be considered as untrusted. What most web pages interact with, however, is the list of name/value pairs. Within a J2EE development environment, a dynamic web page receives the input values as Java strings by calling standard methods provided by the Servlet[8] or JSP[9] container. The Java strings can be stored (and used later) and/or used to form an HTML page as the response to the request. Problem arises when the input string values contain characters that are considered special (markup character) under the HTML specification.

For example, suppose a Hello servlet takes a username input and produces an HTML page that prints the string "Hello" followed by the username:

```
String username = request.getParameter("username");
response.getWriter().println("<html> Hello "+username+"</html>");
```

If the username is "`foo`", the following HTML is sent to the browser:

```
<html>
Hello foo
</html>
```

However, if username is "`foo<b>`", the following is sent:

```
<html>
Hello foo<b>
</html>
```

The sub string "<b>" as part of the username will not be displayed as it is treated as an HTML tag.

With this observation, a malicious user can produce an input such as "`foo <script> ... </script>`", the resulting HTML would be:

```
<html>
Hello foo <script> ... </script>
</html>
```

When a browser receives the HTML, the browser will try to execute the scripts between the script tags.

Can a malicious user make use of this flaw to attack the web site or other users? Attacking the web site directly is not possible using this exploit. However, under the following situations, other users can be indirectly attacked.

Suppose user A's input is first saved by a web site, and later used to form an HTML page to be viewed by user B, the above described flaw in the web site creates a channel for A's script to run in B's browser (since B trusts the web site, B may have allowed scripts embedded in the web site's pages to run in B's browser). Examples of this scenario include user registration (where user entered information is saved and later viewed by site administrators), discussion board (where one user's posting is viewed by others), logging systems (where user inputs are logged and later viewed by administrators).

For pages that do not store user inputs but instead use the inputs to form the HTML response (such as the Hello servlet example), A can trick B into visiting a carefully crafted link pointing to the flawed server page, for example:

```
http://<host>:<port>/servlet/hello?username=foo<script>…</script>
```

When B clicks on the link, the script input embedded in the link again gets executed in B's browser. If one is familiar with the HTML language, one can construct even cleverer attacks along this line so that B is not even aware of the fact that the flawed site is visited. By luring the victim to follow a link or visit an unrelated page, A can execute this attack.

Before discussing prevention techniques, it is useful to understand the potential impact of this type of attack and the ease with which the attack can be launched. Steven Cook's *A Web Developer's Guide to Cross-Site Scripting*[10] gives a good overview of what an XSS attack can do. In summary, from the most serious to the least, the attack can be used to launch session hijacking attacks by stealing other users' cookies set by the flawed web site. The attacker can do almost anything the legitimate users can perform on the affected web site with hijacked sessions. The attack can be used to steal other user's sensitive information by modifying form-posting target, or to modify other form post data. The attack can also be used to create nuisance in victims' browser windows, such as hanging or closing the windows. It should be noted that even though this attack does not directly target a web application's operation, the web application is used as a launch pad for attacking legitimate users of the web site. Web application developers therefore bear the responsibility of preventing the XSS vulnerability in their applications.

Without a malicious user understanding the internal works of a web site, how easy it is for that user to discover a cross-site scripting flaw that can be used to launch an attack? Given the vast number of dynamic pages existing, there are

certainly many opportunities for the attacker. Given the interactive nature of most web applications, the attacker can probe a target by trying different inputs then observing the response from the application and adjusting the attack strings. While it can be time-consuming to try each dynamic page one by one, with the maturing of automated penetration tools for this type of attack[11], we can expect the job of locating a vulnerable page can only become easier.

The automated tools can systematically crawl a web site, while attempting to inject commonly used attack patterns to pages encountered. This helps to narrow down to a few suspicious pages quickly. The tools may require a user to be a legitimate user of the target site (i.e. have a username and password). This can be achieved as many sites today offer self-service features and allow for self-registration. Even if the attacker cannot become a user of a web site, there may be public pages (i.e. pages that do not require authentication) and/or error pages on the site that are susceptible to the XSS attack.

When an attacker discovers a flawed page, is it easy to craft an attack without knowing the internal page logic? This depends on the kind of attack intended. The session hijacking attack (via stolen cookie) and most nuisance attacks do not depend on what a specific page does. Any page vulnerable to XSS attacks can be used to launch them. Certain form manipulation attacks may require better luck at finding the right page.

The crafting of the exact attack string may seem to be a difficult task without understanding the page source code. To remain undetected, the attacker would also like to make the attack as unnoticeable as possible. This can involve not altering the page appearance in any suspicious manner (note that for certain clever attack methods, the victim is not even aware of the fact that a target page is visited, in these cases, the attacker can worry less about not altering the page appearance). In many case, it is not as challenging as it appears to meet these requirements. As mentioned above, an attacker can fine-tune the attack string through interacting with the web application before launching the actual attack.

The first thing an attacker needs to know is usually where the input value is placed in the HTML document the server generates. This can be found out by performing “view source” on the HTML document from the browser. The most common places are in an HTML block (such as the Hello servlet example), or as an attribute value in a tag.

In the first case, the attacker can directly use a script element as the input. Since the script tag is not rendered, it will not alter the page appearance.

In the second case, the attacker often needs to break out of the enclosing attribute value and tag context first. Since attribute values are delimited by double quotes, or single quotes or white spaces, and tags start with ‘<’, an attacker can try inputs that contain script elements but pre-pending them with “>

or '> or whitespace followed by > to break out of the tag first. This is the reason that may automated penetration tools use test inputs that start with "> and '>.

Given the above discussion of the potential damage cross-site scripting causes and the effort required launching an attack, it is reasonable to assume that XSS is a legitimate security threat that should be addressed by web application developers.

Prevention method I

The simplest (and perhaps most performant) form of prevention for this type of attack is to restrict the valid input to be free of characters that have special meanings under the HTML specification[12]. For example, if the value of a user input should be a number, and is validated by the web application as such, we are sure that it cannot be used to launch a cross-site scripting attack.

A common problem in software development is that developers tend to give too much freedom in terms of what values an input can take. Does an input value have to allow for characters such as < and double/single quotes? Many developers ignore such issues at design time or opt for unnecessary flexibility. A reasonably restrictive input set can often greatly simplify a program.

Prevention method II

If it is infeasible to restrict the content of the input, another effective method is to encode/escape the user input on output (i.e. when an application uses the user input to generate an HTML page). Under most circumstances, user inputs are meant to be treated as plain text, without any markups.

The encoding method is discussed in 5. A few points in applying this method in a J2EE environment follow.

The first point is performance. The encoding method requires transferring a string into another where all occurrences of HTML special characters in the original string be replaced with their entity representation (e.g. replace < with `&#60;`). If a lot of the encoding is needed in each generated HTML page, care should be taken to make sure that the encoding method is performant (Java string manipulations can be slow if not coded properly).

The second point is regarding charset. The CERT advisory in 5 mentions that for the encoding method to be effective, it is important to have the correct charset set in the HTTP response header. In servlet or JSP development, you can do the following respectively to set the charset:

```
response.setContentType("text/html; charset=...");
<%@ page contentType="text/raw;charset=..." %>
```

You need to perform this operation before any HTML content is sent to the browser.

Why is setting the charset important? We first explain in more details the charset related portion of HTML, HTTP[13] and servlet/JSP specifications. This is followed by an example of what can happen if charset is not set.

To send data to a browser via the Internet, a web server needs to convert characters into bytes. When the browser receives the data, it needs to convert the bytes back into characters before HTML parsing/rendering can be done. The charset information specifies how the conversion is done. HTTP protocol allows for a "Content-Type" header, which can be used by the web server to communicate to the browser the charset the server used.

In a J2EE environment, if you have used the above-suggested method for servlet or JSP to set the charset, the servlet or JSP container takes care of setting the HTTP Content-Type header using the charset you specified. Furthermore, when you use the PrintWriter object obtained by calling `response.getWriter()` method in a servlet container, or the out object in a JSP container to write HTML strings out to browsers, the servlet or JSP container converts strings into bytes using the same charset you specified.

Each step described above is essential for the encoding method to be effective to prevent XSS attacks. Encoding ensures that any user input in the generated HTML string is free of special markup characters. Each one of the special markup characters corresponds to one or more special byte sequences under a particular charset. Therefore the bytes generated from the encoded user input are also free of those special byte sequences. When a browser uses the same charset to convert these bytes into characters, the browser will not encounter any special markup characters in them.

When charset is not specified or is not correctly specified, a browser may use a charset other than the one used by the server to convert the bytes received. Since different charsets may use different byte sequences to represent the same character, the browser may end up converting the bytes into a character string that contains special characters. This defeats the purpose of server side encoding of user inputs.

Here is an example illustrating a case where the browser and the server disagree on the charset when the content header is not set. What happens when you create a JSP page that contained the following?

```
<%@ page contentType="text/html" %>
<HTML>
<HEAD>
<META http-equiv="Content-Type" content="text/html; charset=UTF-7">
<TITLE>HTML SAMPLE</TITLE>
</HEAD>
```

```
<body>
<+AHM-+AGM-+AHI-+AGk-+AHA-+AHQ->alert("hello");
<+AC8-+AHM-+AGM-+AHI-+AGk-+AHA-+AHQ->
<P>This is a sample HTML page
</body>
</html>
```

Since the charset information is not set within the jsp, the charset in the http response header is not set. In this example, the JSP container will use a default charset to convert the strings to bytes (usually ISO-8859-1). However, when browsers such as the Internet Explorer receive the http response without the charset header, they will attempt other means to figure the charset to use to convert the bytes into characters. In this case, browsers will look at the meta tag in the document and try to interpret the rest of the bytes in UTF-7 encoding. Since under UTF-7, `<+AHM-+AGM-+AHI-+AGk-+AHA-+AHQ->` is an alternative representation for `<script>`, and `<+AC8-+AHM-+AGM-+AHI-+AGk-+AHA-+AHQ->` for `</script>`, the scripts in the script element will be executed. This will not happen if the server and the browser used the same charset to do the conversion between characters and bytes.

The last point is that if an input value appears inside a javascript in an attribute value (e.g. the value of the `onclick` attribute in `<a onclick=...>` can contain javascript), it is necessary to apply both javascript escaping as well as HTML encoding/escaping. The order of escaping should be javascript first, and then HTML, since the browser first performs HTML parsing, and then invokes a scripting engine to process the scripts.

Scenario III

The most complex situation arrives when a web application allows users to enter "safe" markup data, but wishes to reject inputs that contain "unsafe" data. For example, it may allow users to enter HTML containing formatting tags only.

Examples of these applications include sites that allow users to send an e-card and an HTML formatted message to a recipient. Or sites that allow users to send HTML formatted email messages.

Before enabling this feature, developers should be aware of the potential complexity involved in providing the feature in a secure manner.

Two possible approaches to support such a feature are:

1. Identify a safe subset of HTML constructs and allow only those in the safe subset in user inputs.
2. Identify an unsafe subset of HTML constructs and disallow those in the unsafe subset in user inputs.

As an example, we exam a few categories of HTML constructs in detail. The first one contains constructs that can introduce scripting content (such as JavaScript or VBScript). As far as XSS is concerned, this is the category that developers need to pay attention to.

The HTML specification tells where in an HTML document scripts can appear. There are three categories in general:

1. In a `script` element, scripts can appear between the start and end script tag. Or, an external script can be referred to by specifying its location as the value of the `src` attribute in the open script tag.
2. Scripts can appear as the attribute value for an intrinsic event attribute. Intrinsic even attributes appear inside tags, these include `onclick`, `onclick`, `ondblclick`, `onmousedown`, `onmouseup`, `onmouseover`, `onmousemove`, `onmouseout`, `onkeypress`, `onkeydown`, `onkeyup`, `onload`, `onunload`, `onfocus`, `onblur`, `onsubmit`, `onreset`, `onselect`, `onchange`, `onAbort`, `onDragDrop`, `onError`, `onMove` and `onResize`.
3. In addition to the two categories above, users can place scripts in an HTML document anywhere the HTML specification allows for a URI type of data. Examples are:
    a. The value of the `href` attribute in an `<a>` tag can be scripts: `<a href="JavaScript:alert('hello');">hyperlink to a javascript function</a>`. The other common attribute that can take URI type of data is `src`.
    b. Scripts can appear inside the value of the style attribute in certain tags: `<input name=myinput style=background:url(javascript:alert('hello'))>`
    c. Scripts can appear inside a style element: `<STYLE type="text/css"> input {background:url(vbscript:alert("hi")) } </STYLE>`

Besides scripts, developers may want to examine the following groups of constructs for security purposes (this is, however, outside the scope of the XSS problem).

One group allows the introduction of other forms of code, this include the applet element, the object element and the embed element. Developers need to decide whether to allow these constructs based on the feature and security requirements.

Form element is another construct that requires careful treatment. If a web site allows users to send forms to another user, can this allow malicious users to trick others into submitting something sensitive? Remember, the form submit target is not visible to end users.

The last group we mention here are <a> and <frame>. Developers need to decide whether to allow users to insert links and frames into the HTML that a server generates. These constructs may seem harmless at first; however there can be security issues. For example, if the URL for the HTML page containing the <a> element contains sensitive information, when a user clicks on the <a> link, that URL will be sent as the "referrer URL" to the site where <a> points to.

What makes this prevention method the most complex of the three is that each of the HTML construct requires careful examination (some constructs may have been obsoleted, but are still being recognized by browsers). Once the safe or unsafe set is determined, each user input needs to be parsed to determine its safety property. The parsing process may need to take into consideration of the implementation specifics of browsers (for example, many browsers ignore the NUL character when parsing). Finally, since the HTML standard is evolving, new constructs can be added. Therefore this method may require on-going maintenance efforts.

Conclusion

Cross-site scripting attack is a legitimate security threat to dynamic web sites. It is regarded as one of the top security flaws existing in today's dynamic sites. With the attack method becoming more mature and automated, and the fact that more dynamic web sites are being set up, we can expect the problem to become worse.

While a number of articles have given examples of the attack, as well as testing, prevention, mitigation methods, this paper attempted to focus on the prevention methods, giving more details especially in J2EE development environments. We hope that the technical details provided will help developers understand and protect their applications against this attack.

Reference

[1] CERT® Coordination Center. "CERT® Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests." Feb 2000. URL: http://www.cert.org/advisories/CA-2000-02.html (4 May 2004)

[2] NEOHAPSIS ARCHIVES. URL: http://archives.neohapsis.com/archives/bugtraq/2000-08/0268.html (4 May 2004)

[3] NEOHAPSIS ARCHIVES. URL: http://archives.neohapsis.com/archives/ntbugtraq/2000-q3/0105.html (4 May 2004)

[4] Microsoft. " Information on Cross-Site Scripting Security Vulnerability." Feb 2000. URL: http://www.microsoft.com/technet/security/news/crssite.mspx (4 May 2004)

[5] CERT® Coordination Center. "Understanding Malicious Content Mitigation for Web Developers." URL: http://www.cert.org/tech_tips/malicious_code_mitigation.html (4 May 2004)

[6] APACHE. "Cross Site Scripting Info: Encoding Examples." URL: http://httpd.apache.org/info/css-security/encoding_examples.html (4 May 2004)
[7] The Open Web Application Security Project. URL: http://www.owasp.org/documentation/topten (4 May 2004)
[8] Sun Microsystems. "Java Servlet Technology" URL: http://java.sun.com/products/servlet/ (4 May 2004)
[9] Sun Microsystems. "JavaServer Pages" URL: http://java.sun.com/products/jsp/download.html#specs (4 May 2004)
[10] Cook, Steven. "A Web Developer's Guide to Cross-Site Scripting." 11 Jan 2003. URL: http://www.sans.org/rr/papers/46/988.pdf (4 May 2004)
[11] Sanctum. AppScan product. URL: http://www.sanctuminc.com/solutions/appscanqa/index.html (4 May 2004)
[12] W3C. "HTML 4.01 Specification." 24 Dec. 1999. URL: http://www.w3.org/TR/html401/ (4 May 2004)
[13] W3C. "Hypertext Transfer Protocol -- HTTP/1.1. " URL: http://www.w3.org/Protocols/rfc2616/rfc2616.html (4 May 2004)