Advanced Cross Site Scripting

by Gavin Zuchlinski
http://libox.net/
10/16/2003

Table of Contents

Introduction

I recently read in an article the incorrect statement that cross site scripting (XSS) can not be exploited if the POST method is used instead of GET, which is completely false. The method used to exploit POST variables may also be modified to allow for more advanced timing attacks which could allow an attacker to gain access to areas that require the user log in to a password protected area. When coupled with social engineering this method becomes an extremely reliable tool for attackers to gain access to secured areas via account hijacking.

In typical cross site scripting the target views a website which contains code inserted into the HTML which was not written by the website designer or administrator. This bypasses the document object model which was intended to protect domain specific cookies (sessions, settings, etc.). In most instances the target is sent a link to a website on the server which the target has a legitimate account and by viewing that website the attackers malicious code is executed (commonly javascript to send the user's cookie to a third party server, in effect stealing their session and their account). This was a quick overview of cross site scripting and a solid foundation is needed before proceeding, my recommended reading is iDefense's XSS article (google.com). The attack presented below in conjunction with iDefense's method of attack automation makes for a very powerful combination.

NOTE (October 19 2003) – Sverre Huseby has brought to my attention that the generalized version attack is not unique, it was discovered first by Jim Fulton (http://www.zope.org/Members/jim/ZopeSecurity/ClientSideTrojan), without my knowledge.

Post Method

Because POST variables are sent separate of the actual website URL a direct attack from the target clicking on the malicious link and directly accessing the server vulnerable to the XSS attack is not possible (as far as I know). This is opposed to a GET request where the variable arguments are stored in the URL, such as http://www.google.com/search?hl=en&q=xss where the variables hl and q are seen in the URL. The implications of variables being sent in this manner are not in the scope of this article, but the POST method sends variables in the HTTP request and is not integrated in the URL such as is the case with GET.

To exploit a web page with a cross site scripting vulnerability via a GET variable a URL in the form of http://vulnerable.com/search?q=<script>alert(document.cookie) </script> is composed. This URL is then sent to the target, upon clicking the URL they are taken to vulnerable.com's handy search engine (not to mention the dual HTML rendering within their site functionality) and the target receives a javascript pop up with their session cookie.

Creating exploits for POST requests are only trivially more difficult, an intermediary web page is needed which will hold code that will force the client web browser in to making the POST request to the vulnerable server. This is trivially done via a form (with method POST and action of the target script) and javascript code which will automatically submit the form on page load. See example code block below.

```
<form method="POST" action="http://vulnerable.com/search"
name="explForm">
<input type=hidden name=q value="<script>alert
(document.cookie)</script>">
</form>
<script language="Javascript">
setTimeout('explForm.submit()', 1);
</script>
```

One millisecond after the page is loaded containing this code the form (completely invisible in the rendered HTML) is submitted. In this case you have a simple search for "<script>alert(document.cookie) </script>" done on vulnerable.com's search engine (and consequently a javascript alert appears because for the sake of this paper, vulnerable.com's search engine is vulnerable to a cross site scripting attack). The above code can be easily changed if the target script

requires variables to be GET, change method="POST" to method="GET". The above code can be placed on a static web page on a web server controlled by the attacker and then the link sent to the target. Another vector to deliver the form and javascript to the target is via a site vulnerable to XSS through a GET request.

In either case above the attacker sends the target the malicious web page, the malicious web page forms the request and the request is sent to the vulnerable server. This advances the classical cross site scripting attack from a single hop (target --> page within vulnerable website containing inserted code) to two hops (target --> intermediary request formulation page --> page within vulnerable website containing inserted code).

Expansion on POST: secure areas

The problem of password protected areas also arises, where a password is required every time the user accesses the website. In many websites which require secure client access the cookie is not persistent to prevent further users on the computer from logging in to the account.

Building upon the code presented above we can circumvent any restrictions and still steal the session cookie for the temporary session. Unfortunately the time window in which attacks can take place in many cases is very small, with the help of iDefense's idea of automating attacks this small time window is no longer an issue. By adding code on the intermediary web page which opens a new window with the login prompt the user may now log in to the secured area (some social engineering might be required in order to force the user to log in). See code below.

```
<form method="POST" action="http://vulnerable.com/search"
name="explForm">
<input type=hidden name=q value="<script>alert
(document.cookie)</script>">
</form>
<script language="Javascript">
window.open("http://vulnerable.com/secure_login");
setTimeout('explForm.submit()', 1000*30);
</script>
```

Note: changes from previous code diplayed in bold

With the intermediary web page still in the background, the form submission may now be timed to allow the user to log in successfully before the exploit is sent. To change the time until the form is submitted change the second argument in the setTimeout function, this is the time in milliseconds until the javascript code in argument one is executed. With the user successfully logged in a child window of the intermediary web page, when the form on the intermediary web page is submitted the form will go directly to the problematic script, malicious code inserted, and the user session may be stolen.

Using an intermediary for exploitation slightly increases the complexity of a successful attack but allows for a high degree of flexibility, any variable that is used on a dynamically created web page which does not sanitize HTML markup is vulnerable to cross site scripting.

Generalized Client Automation

Generalizing on the above technique brings to light another, and in some cases a very serious, vulnerability. The proposed technique allows an attacker to fill out forms with data they specify and submit them automatically under the context of the client. Any forms which accept data from the client, assuming they in fact inputted the data they are submitting, are vulnerable.

This arises when the form itself is dependent only on static or predictable information (information given to a third party site such as referrer can help in prediction). Using the method of exploitation presented above, client automation of form submission is a trivial task.

```
<form method="POST"
action="http://vulnerable.com/changeMailSettings" name="f">
<input type=hidden name=reply_to value="attacker@h4x.com">
<input type=hidden name=signature value="<a
href=http://h4x.com/exploit.htm>Click here</a> for a free
computer security test, trust me, I used it and was
amazed!">
</form>
<script language="Javascript">
f.submit();
</script>
```

An interesting use of this would be the creation of a webmail signature virus. Using the techniques presented above the attacker could compose a web page that when visited would automate the the form which changes the signature sent out on emails to contain the link to the malicious page itself. Every time a user “infected” with the signature virus would send an email unknowingly they would also send along text and a link persuading the next victim to also click it, and become infected. Easy automated spamming? Yes.

Hotmail and Yahoo! Mail have both been tested for this vulnerability and they are secure against it, however each appear to have combated the flaw in very different ways. Hotmail uses a simple referrer check, if the referrer is not from an authorized Hotmail page the user is sent directly to a login page. Yahoo enacted a very novel approach to fix the problem, on each form there is a hidden value named “.crumb” which is related to the cookie. All protection against this flaw lies within the crumb, if the crumb can be predicted without the cookie then Yahoo is vulnerable to this flaw.

Prevention

Because the generalized client automation attack is very simple at the server end (ideally the server views only a legitimate request by the client) it is somewhat more difficult to prevent. Due to the fact that the client forms the request at their browser HTTP Referrer headers can be trusted and should be validated to ensure they come from an internal script inside the system. Referrer checking assumes however that the attacker can not insert arbitrary HTML in to any of the trusted scripts, though such attack would be considered cross site scripting and separate from this.

At the very minimum to protect against cross site scripting attacks user input must be stripped of any potentially dangerous characters such as < > “ &. As any conscientious security professional would do, I must preach the importance of the whitelisting approach over blacklisting; in whitelisting only explicitly allowed characters are permitted in the input. It appears that all security vulnerabilities stem from user input, “Hello world” can not be exploited unless the attacker can manage in some form to input data. This should lead us to believe that it is trivial to ensure security (in a large amount of cases of cases, but not all) by validating user input to a strict form. Regular expressions are extremely powerful for the task of whitelisting characters and validating that data does in fact conform to the form standards (include length constraints also in the concept of form). Once data is validated to a set of criteria security analysis is purely creative thinking of how the criteria may be managed to let through specific items it should not.

Another option to aid in the prevention of security flaws is a project I am affiliated with, currently code named Nirvana. This project is devoted to creation of user input filters and validation function to help developers create secure code faster. The project page is now housed at http://libox.net/sanitize.php but will soon be moving to http://www.owasp.org.

My home on the web is http://libox.net/, the most current version of this document may be found there.