DOM Based Cross Site Scripting or XSS of the Third Kind

A look at an overlooked flavor of XSS

By Amit Klein (aksecurity (at) hotpop (dot) com)
Version 0.2.8
Last Modified: 7/4/2005

[TEXT] size: 22k (MD5 SUM: 2285b3e1e8c471ce4d966958213e47c2)
[HTML] size: 42k (MD5 SUM: 237cae7486922ce7c7580ffee3fcca30)

Summary

We all know what Cross Site Scripting (XSS) is, right? It's that vulnerability wherein one sends malicious data (typically HTML stuff with Javascript code in it) that is echoed back later by the application in an HTML context of some sort, and the Javascript code gets executed. Well, wrong. There's a kind of XSS which does not match this description, at least not in some fundamental properties. The XSS attacks described above are either "non-persistent"/"reflected" (i.e. the malicious data is embedded in the page that is returned to the browser immediately following the request) or "persistent"/"stored" (in which case the malicious data is returned at some later time). But there's also a third kind of XSS attacks - the ones that do not rely on sending the malicious data to the server in the first place! While this seems almost contradictory to the definition or to common sense, there are, in fact, two well described examples for such attacks. This technical note discusses the third kind of XSS, dubbed "DOM Based XSS". No claim is made to novelty in the attacks themselves, of course, but rather, the innovation in this write-up is about noticing that these belong to a different flavor, and that flavor is interesting and important.

Application developers and owners need to understand DOM Based XSS, as it represents a threat to the web application, which has different preconditions than standard XSS. As such, there are many web applications on the Internet that are vulnerable to DOM Based XSS, yet when tested for (standard) XSS, are demonstrated to be "not vulnerable". Developers and site maintainers (and auditors) need to familiarize themselves with techniques to detect DOM Based XSS vulnerabilities, as well as with techniques to defend against them, both therewhich are different than the ones applicable for standard XSS.

Introduction

The reader is assumed to possess basic knowledge of XSS ([1], [2], [3], [4], [8]). XSS is typically categorized into "non-persistent" and "persistent" ([3], "reflected" and "stored" accordingly, as defined in [4]). "Non-persistent" means that the malicious (Javascript) payload is echoed by the server in an immediate response to an HTTP request from the victim. "Persistent" means that the payload is stored by the system, and may later be embedded by the vulnerable system in an HTML page provided to a victim. As mentioned in the summary, this categorization assumes that a fundamental property of XSS is having the malicious payload move from the browser to the server and back to the same (in non-persistent XSS) or any (in persistent XSS) browser. This paper points out that this is a misconception. While there are not many counterexamples in the wild, the mere existence of XSS attacks which do not rely on the payload embedded by the server in some response page, is of importance as it has a significant impact on detection and protection methods. This is discussed in the document.

Example and Discussion

Before describing the basic scenario, it is important to stress that the techniques outlined here were already demonstrated in public (e.g. [5], [6] and [7]). As such, it is not claimed that the below are new techniques (although perhaps some of the evasion techniques are).

The prerequisite is for the vulnerable site to have an HTML page that uses data from the `document.location` or `document.URL` or `document.referrer` (or any various other objects which the attacker can influence) in an insecure manner.

NOTE for readers unfamiliar with those Javascript objects: when Javascript is executed at the browser, the browser provides the Javascript code with several objects that represent the DOM (Document Object Model). The `document` object is chief among those objects, and it represents most of the page's properties, as experienced by the browser. This `document` object contains many sub-objects, such as `location`, `URL` and `referrer`. These are populated by the browser according to the browser's point of view (this is significant, as we'll see later with the fragments). So, `document.URL` and `document.location` are populated with the URL of the page, as the browser understands it. Notice that these objects are not extracted of the HTML body - they do not appear in the page data. The `document` object does contain a `body` object that is a representation of the parsed HTML.

It is not uncommon to find an application HTML page containing Javascript code that parses the URL line (by accessing `document.URL` or `document.location`) and performs some client side logic according to it. The below is an example to such logic.

In analogy to the example in [2] (and as an essentially identical scenario to the one in [7]), consider, for example, the following HTML page (let's say this is the content of `http://www.vulnerable.site/welcome.html`):

```
<HTML>
<TITLE>Welcome!</TITLE>
Hi
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<BR>
Welcome to our system
…
</HTML>
```

Normally, this HTML page would be used for welcoming the user, e.g.:

```
http://www.vulnerable.site/welcome.html?name=Joe
```

However, a request such as:

```
http://www.vulnerable.site/welcome.html?name=
<script>alert(document.cookie)</script>
```

would result in an XSS condition. Let's see why: the victim's browser receives this link, sends an HTTP request to `www.vulnerable.site`, and receives the above (static!) HTML page. The victim's browser then starts parsing this HTML into DOM. The DOM contains an object called document, which contains a property called URL, and this property is populated with the URL of the current page, as part of DOM creation. When the parser arrives to the Javascript code, it executes it and it modifies the raw HTML of the page. In this case, the code references `document.URL`, and so, a part of this string is embedded at parsing time in the HTML, which is then immediately parsed and the Javascript code found (`alert(…)`) is executed in the context of the same page, hence the XSS condition.

Notes:
1. The malicious payload was not embedded in the raw HTML page at any time (unlike the other flavors of XSS).

2. This exploit only works if the browser does not modify the URL characters. Mozilla automatically encodes < and > (into %3C and %3E, respectively) in the `document.URL` when the URL is not directly typed at the address bar, and therefore it is not vulnerable to the attack as shown in the example. It is vulnerable to attacks if < and > are not needed (in raw form). Microsoft Internet Explorer 6.0 does not encode < and >, and is therefore vulnerable to the attack as-is.

Of course, embedding in the HTML directly is just one attack mount point, there are various scenarios that do not require < and >, and therefore Mozilla in general is not immune from this attack.

Evading standard detection and prevention technologies

In the above example, it may be argued that still, the payload did arrive to the server (in the query part of the HTTP request), and so it can be detected just like any other XSS attack. But even that can be taken care of. Consider the following attack:

http://www.vulnerable.site/welcome.html#name=<script>alert(document.cookie)<script>

Notice the number sign (#) right after the file name. It tells the browser that everything beyond it is a fragment, i.e. not part of the query. Microsoft Internet Explorer (6.0) and Mozilla do not send the fragment to the server, and therefore, the server would see the equivalent of `http://www.vulnerable.site/welcome.html`, so the payload would not even be seen by the server. We see, therefore, that this evasion technique causes the major browsers not to send the malicious payload to the server.

Sometimes, it's impossible to completely hide the payload: in [5] and [6], the malicious payload is part of the username, in a URL that looks like `http://username@host/`. The browser, in such case, sends a request with Authorization header containing the username (the malicious payload), and thus, the payload does arrive to the server (Base64 encoded - so IDS/IPS would need to decode this data first in order to observe the attack). Still, the server is not required to embed this payload in order for the XSS condition to occur.

Obviously, in situations where the payload can be completely hidden, online detection (IDS) and prevention (IPS, web application firewalls) products cannot fully defend against this attack, assuming the vulnerable script can indeed be invoked from a known location. Even if the payload has to be sent to the server, in many cases it can be crafted in such way to avoid being detected, e.g. if a specific parameter is protected (e.g. the name parameter in the above example), then a slight variation of the attack may succeed:

```
http://www.vulnerable.site/welcome.html?notname=<script>(document.cookie)</script>
```

A more strict security policy would require that the name parameter be sent (to avoid the above tricks with names and number sign). We can therefore send this:

```
http://www.vulnerable.site/welcome.html?notname=
<script>alert(document.cookie)<script>&name=Joe
```

If the policy restricts the additional parameter name (e.g. to `foobar`), then the following variant would succeed:

```
http://www.vulnerable.site/welcome.html?foobar=
name=<script>alert(document.cookie)<script>&name=Joe
```

Note that the ignored parameter (`foobar`) must come first, and it contains the payload in its value.

The scenario in [7] is even better from the attacker's perspective, since the full `document.location` is written to the HTML page (the Javascript code does not scan for a specific parameter name). Therefore, the attacker can completely hide the payload e.g. by sending:

```
/attachment.cgi?id=&action=
foobar#<script>alert(document.cookie)</script>
```

Even if the payload is inspected by the server, protection can be guaranteed only if the request in its fullness is denied, or if the response is replaced with some error text. Consider [5] and [6] again, if the Authorization header is simply removed by an intermediate protection system, it has no effect as long as the original page is returned. Likewise, any attempt to sanitize the data on the server, either by removing the offending characters or by encoding them, is ineffective against this attack.

In the case of `document.referrer` , the payload is sent to the server through the Referer header. However, if the user's browser, or an intermediate device eliminates this header, then there's no trace of the attack - it may go completely unnoticed.

To generalize, traditional methods of:

1. HTML encoding output data at the server side
2. Removing/encoding offending input data at the server side

Do not work well against DOM Based XSS.

Regarding automatic vulnerability assessment by way of fault injection (sometimes called fuzzing) won't work, since products that use this technology typically evaluate the results according to whether the injected data is present in the response page or not (rather than execute the client side code in a browser context and observe the runtime effects). However, if a product is able to statically analyze a Javascript found in a page, then it may point out suspicious patterns (see below). And of course, if the product can execute the Javascript (and correctly populating the DOM objects), or simulate such execution, then it can detect this attack.

Manual vulnerability assessment using a browser would work because the browser would execute the client side (Javascript) code. Of course, a vulnerability assessment product may adopt this kind of technology and execute client side code to inspect the runtime effects.

Effective defenses

1. Avoiding client side document rewriting, redirection, or other sensitive actions, using client side data. Most of these effects can be achieved by using dynamic pages (server side).

2. Analyzing and hardening the client side (Javascript) code. Reference to DOM objects that may be influenced by the user (attacker) should be inspected, including (but not limited to):

- `document.URL`
- `document.URLUnencoded`
- `document.location (and many of its properties)`
- `document.referrer`
- `window.location (and many of its properties)`

Note that a document object property or a window object property may be referenced syntactically in many ways - explicitly (e.g. `window.location`), implicitly (e.g. `location`), or via obtaining a handle to a window and using it (e.g. `handle_to_some_window.location`).

Special attention should be given to scenarios wherein the DOM is modified, either explicitly or potentially, either via raw access to the HTML or via access to the DOM itself, e.g. (by no means an exhaustive list, there are probably various browser extensions):

- Write raw HTML, e.g.:
  - `document.write(…)`
  - `document.writeln(…)`
  - `document.body.innerHtml=…`
- Directly modifying the DOM (including DHTML events), e.g.:
  - `document.forms[0].action=…` (and various other collections)
  - `document.attachEvent(…)`
  - `document.create…(…)`
  - `document.execCommand(…)`
  - `document.body. …` (accessing the DOM through the body object)
  - `window.attachEvent(…)`
- Replacing the document URL, e.g.:
  - `document.location=…` (and assigning to location's href, host and hostname)
  - `document.location.hostname=…`
  - `document.location.replace(…)`
  - `document.location.assign(…)`
  - `document.URL=…`
  - `window.navigate(…)`
- Opening/modifying a window, e.g.:
  - `document.open(…)`
  - `window.open(…)`
  - `window.location.href=…` (and assigning to location's href, host and hostname)
- Directly executing script, e.g.:
  - `eval(…)`
  - `window.execScript(…)`
  - `window.setInterval(…)`
  - `window.setTimeout(…)`

To continue the above example, an effective defense can be replacing the original script part with the following code, which verifies that the string written to the HTML page consists of alphanumeric characters only:

```
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
var name=document.URL.substring(pos,document.URL.length);
if (name.match(/^[a-zA-Z0-9]$/))
{
   document.write(name);
}
else
{
    window.alert("Security error");
}
</SCRIPT>
```

Such functionality can (and perhaps should) be provided through a generic library for sanitation of data (i.e. a set of Javascript functions that perform input validation and/or sanitation). The downside is that the security logic is exposed to the attackers - it is embedded in the HTML code. This makes it easier to understand and to attack it. While in the above example, the situation is very simple, in more complex scenarios wherein the security checks are less than perfect, this may come to play.

3. Employing a very strict IPS policy in which, for example, page welcome.html is expected to receive a one only parameter named “name”, whose content is inspected, and any irregularity (including excessive parameters or no parameters) results in not serving the original page, likewise with any other violation (such as an Authorization header or Referer header containing problematic data), the original content must not be served. And in some cases, even this cannot guarantee that an attack will be thwarted.

A note about redirection vulnerabilities

The above discussion is on XSS, yet in many cases, merely using a client side script to (insecurely) redirect the browser to another location is considered vulnerability in itself. In such cases, the above techniques and observations still apply.

Conclusion

While most XSS attacks described in public do indeed depend on the server physically embedding user data into the response HTML pages, there are XSS attacks that do not rely on server side embedding of the data. This has material significance when discussing ways to detect and prevent XSS. To date, almost all detection and prevention techniques discussed in public assume that XSS implies that the server receives malicious user input and embeds it in an HTML page. Since this assumption doesn’t hold (or only very partially holds) for the XSS attacks described in this paper, many of the techniques fail to detect and prevent this kind of attacks.

The XSS attacks that rely on server side embedding of user data are categorized into “non-persistent” (or “reflected”) and “persistent” (or “stored”). It is thus suggested that the third kind of XSS, the one that does not rely on server side embedding, be named “DOM Based XSS”.

Here is a comparison between standard XSS and DOM Based XSS:

	Standard XSS	DOM Based XSS
Root cause	Insecure embedding of client input in HTML outbound page	Insecure reference and use (in a client side code) of DOM objects that are not fully controlled by the server provided page
Owner	Web developer (CGI)	Web developer (HTML)
Page nature	Dynamic only (CGI script)	Typically static (HTML), but not necessarily.
Vulnerability Detection	• Manual Fault injection • Automatic Fault Injection • Code Review (need access to the page source)	• Manual Fault Injection • Code Review (can be done remotely!)
Attack detection	• Web server logs • Online attack detection tools	If evasion techniques are applicable and used - no server side detection is

	(IDS, IPS, web application firewalls)	possible
Effective defense	• Data validation at the server side • Attack prevention utilities/tools (IPS, application firewalls)	• Data validation at the client side (in Javascript) • Alternative server side logic

References

Note: the URLs below are up to date at the time of writing (July 4th, 2005). Some of these materials are live documents, and as such may be updated to reflect the insights of this paper.

[1] "CERT Advisory CA-2000-02 - Malicious HTML Tags Embedded in Client Web Requests", CERT, February 2nd, 2000

http://www.cert.org/advisories/CA-2000-02.html

[2] "Cross Site Scripting Explained", Amit Klein, June 2002

http://crypto.stanford.edu/cs155/CSS.pdf

[3] "Cross-Site Scripting", Web Application Security Consortium, February 23rd, 2004

http://www.webappsec.org/projects/threat/classes/cross-site_scripting.shtml

[4] "Cross Site Scripting (XSS) Flaws", The OWASP Foundation, updated 2004
http://www.owasp.org/documentation/topten/a4.html

[5] "Thor Larholm security advisory TL#001 (IIS allows universal CrossSiteScripting)", Thor Larholm, April 10th, 2002

http://www.cgisecurity.com/archive/webservers/iis_xss_4_5_and_5.1.txt

(see also Microsoft Security Bulletin MS02-018
http://www.microsoft.com/technet/security/bulletin/MS02-018.mspx)

[6] "ISA Server Error Page Cross Site Scripting", Brett Moore, July 16th, 2003
http://www.security-assessment.com/Advisories/ISA%20XSS%20Advisory.pdf

(see also Microsoft Security Bulletin MS03-028
http://www.microsoft.com/technet/security/bulletin/MS03-028.mspx and a more elaborate description in "Microsoft ISA Server HTTP error handler XSS", Thor Larholm, July 16th, 2003 http://www.securityfocus.com/archive/1/329273)

[7] "Bugzilla Bug 272620 - XSS vulnerability in internal error messages", reported by Michael Krax, December 23rd, 2004

https://bugzilla.mozilla.org/show_bug.cgi?id=272620

[8] “The Cross Site Scripting FAQ”, Robert Auger, May 2002 (revised August 2003)

http://www.cgisecurity.com/articles/xss-faq.shtml

About the author

Amit Klein is a renowned web application security researcher. Mr. Klein has written many research papers on various web application technologies--from HTTP to XML, SOAP and web services--and covered many topics--HTTP request smuggling, insecure indexing, blind XPath injection, HTTP response splitting, securing .NET web applications, cross site scripting, cookie poisoning and more. His works have been published in Dr. Dobb's Journal, SC Magazine, ISSA journal, and IT Audit journal; have been presented at SANS and CERT conferences; and are used and referenced in many academic syllabi.

Mr. Klein is a WASC (Web Application Security Consortium) member.