# Hacking Web 2.0 JavaScript - Reverse Engineering, Discovery and Revelations


## Abstract

Traditionally a large number of applications were carried out without the intervention of global networks like the Internet. But now, as the Web 2.0 era is emerging at an increasingly fast rate today and is here to stay, these applications are becoming increasingly dependent on the internet as a foundation platform. As the application domain increases worldwide, the variety in the kind of web content also increases and rises above mere traditional HTML. The kind of enhancements brought about in HTML pages, as viewed by a client, are introduced by technologies such as JavaScript, Flash and Silverlight. Since, these applications are widely growing and becoming crucial, here the intention is to throw light on the methods to look for security loopholes such as XSS (Cross-Site Scripting) in JavaScript, specific to the Web 2.0 implementations of the same which consume information from the un-trusted sources. The methods described pertain to static as well as dynamic analysis. Tools that have been employed in this paper are

- Static Code Analysis of JavaScript by AppCodeScan ( http:// blueinfy.com/appcodeaudit.html)
- Dynamic Debugging and Analysis by using firebug with DOM context (http://getfirebug.com/).


Blueinfy AppSecLabs


***Rishita Anubhai (rishita@blueinfy.net)***

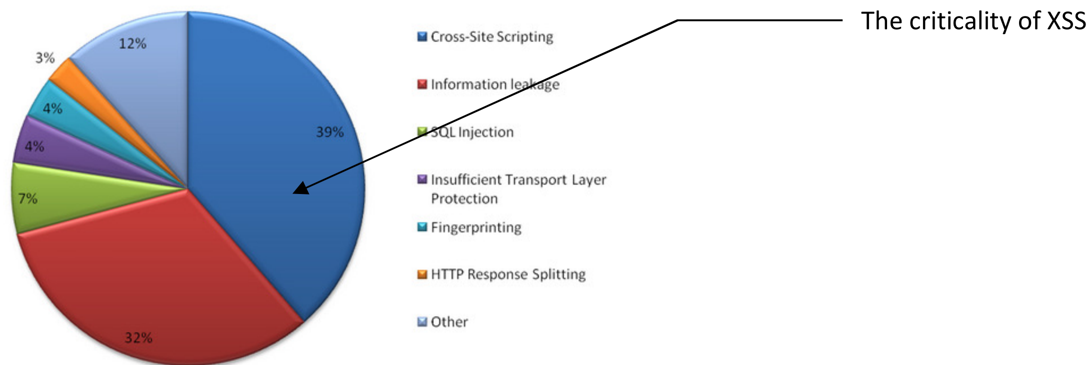
## Problem Domain

JavaScript as we know is a client side technology that uses scripting on the client side to process certain elements of the HTML page while it is rendered on the browser with DOM context – the client. In such a case,

- All the client side code files are available for the users to inspect and look out for validation loopholes and any other business logic residing.
- The malicious attackers have a chance to inject client side code snippets in their own favour, since the <script> tags are permissible in HTML pages with the coming of JavaScript.
- With reference to the Web 2.0 applications, JavaScript contains a lot of information and business logic in specific. This logic can be reverse engineered by an attacker.
- Web 2.0 applications are using the Document Object Model (DOM) extensively. As a result, it becomes easier for an attacker to exploit weakly implemented DOM calls across client side logic. For example, eval() on an un-trusted stream.
- XMLHttpRequests (XHR) Object also makes hidden calls at the back-end. These calls can be discovered and exploited by any attacker. We have seen popular applications like Twitter or Facebook that were compromised by exploiting XHR calls in this manner.

Of the many web security attacks that can be used as a result of this, the recent statistics show the following for the distribution with regard to the type of attacks:
(http://projects.webappsec.org/Web-Application-Security-Statistics)

### Percent of vulnerabilities out of total number of vulnerabilities




Since the one of the easiest exploitations of JavaScript occurs via the XSS attack it has been taken up as the focus hereafter. It is imperative to observe the trend as well; we are seeing rise of DOM based XSS in Web 2.0 application context. The synoptic view for the same is as follows:

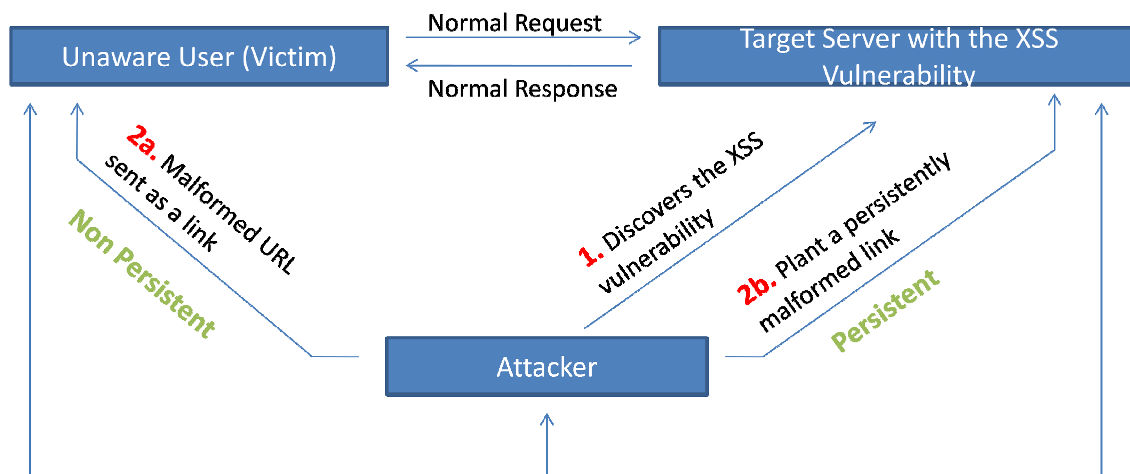
## XSS: Cross-Site Scripting

XSS is basically a scenario where the 'trust on the user, of the hosting webpage is exploited'. This means that client-side code snippets will be injected into the places that the host of the page least expects and will be used for maliciously.

The three major categories can be summarized as described on the next page but we will focus on DOM based XSS during this particular paper:

| Type | Fundamental Concept | | Example Scenario |
|---|---|---|---|
| | **Vulnerability** | **Exploit** | |
| Reflected (Non Persistent) | Any user input enabled field's value is directly reflected back into the response page without validation. | A script tag that includes JavaScript code snippet can hence be injected in this field. Thus, when response is generated the code gets executed.<br>Using this category, URLs could be malformed to pass parameter values with such codes and used by third party attackers for vulnerable sites and corresponding user victims. | A field which takes for input, a keyword for searching and generates the result page with the string '<keyword-here> search results:'. Here if along with the keyword the user also adds: **'<script>alert("xss");</script>**'.<br>Then when the field value is extracted and duplicated on the response page, this script also becomes a part of the response and gets auto-executed. |
| Stored (Persistent) | The HTML links posted on message-boards, blogs are tailored in a manner that already JavaScript code. These will be permanently stored on the server of this message-board or blog site. | The malformed link being permanently stored on the server, whenever visited by any user will cause the code to get executed each time.<br>No individual targets are required. | Consider the following link on a page:<br>**<a href=http://xyz.com/home.htm?name=3;<script>alert("hi"); </script>">>**<br>Thus, whenever clicked, the unaware user visits the page home.htm but also causes the code to be executed which could extract information and pass it on to the attacker |
| DOM based | Here, the response page is not altered. The Document-Object-Model aspects of the page are accessed and caused to behave unexpectedly to each user based on the DOM modifications that have been maliciously caused. Various different calls like document.write or eval can be exploited. | | The document.location object could contain a string such as **http://xyz.com/home.htm?default=<script>alert(document.cookie); </script>.**<br>Since this might not expect a code snippet it will render the entire page here and the script too is rendered and executed alongside. |

## Diagrammatic Representation of the Scenario




**3.** Now all communication via the malformed Request-Response Pairs happens as directed by the attacker's injected code snippets. Example: He can ask for the victim's cookies.

# Approach and Tools

The approaches to analyse the JavaScript loopholes as aforementioned can be categorized into two distinct types:

## Static Code Analysis

In this approach, we will be downloading all the JavaScript files along with the HTML source code of the target page to be audited for such loopholes and analyse it without running it. This means, that without execution of the source and interference by supplying runtime data, we shall merely examine the call-return hierarchy of the code and look for loopholes such as slack validation during XHR Requests and others that will be proposed at a later point.

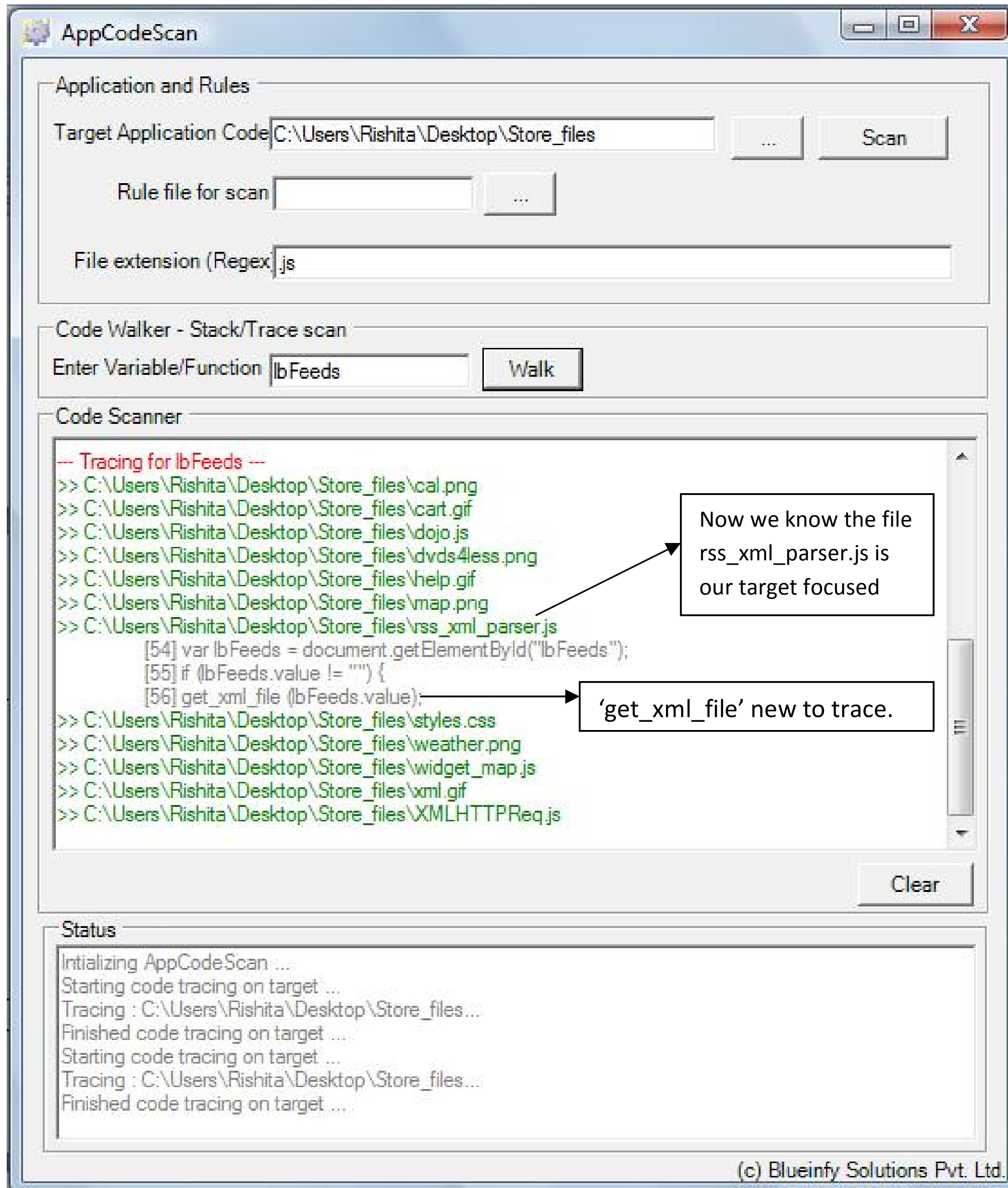
Since the code can be very lengthy to go over it manually the tool that will be used in the demonstration is the AppCodeScan Tool, a tool put forth by Blueinfy for the purpose of tracking and walking over functions via calls. This will be used for our purpose here by putting the suspicious start point in the 'Trace' field to start with and then analysing the results.

The detailed demonstrations will be accompanied by screenshots now for a particular example. We download the source page for the following page which has widgets. Seeing the html page the following is notable:

This triggers us to check get_rss_feed and lbFeeds

```
<DIV align=center><SELECT id=lbFeeds
onchange=get_rss_feed(); name=lbFeeds> <OPTION
selected
value=rss/proxy.aspx?url=http://rss.cnn.com/rss/cnn_topstories.rss>CNN
business</OPTION> <OPTION
value=rss/proxy.aspx?url=http://asp.usatoday.com/marketing/rss/rsstrans.aspx?feedId=news1>USA
today business</OPTION> <OPTION
value=rss/proxy.aspx?url=http://192.168.150.23/rss/news.xml>Trade
news</OPTION></SELECT> <INPUT id=cbDetails
onclick='format ("content", last_xml_response);'
type=hidden name=hidden> </DIV></FORM></TD></TR>
```

Therefore we trace the function 'get_rss_feed' & 'lbFeeds' in the .js files loaded from the page.




Thus now we walk for 'get_xml_file'. The result in our target focus file gives:

```
>> C:\Users\Rishita\Desktop\Store_files\rss_xml_parser.js
        [24] function get_xml_file (url) {
        [56] get_xml_file (lbFeeds.value);
```

Seeing this, we take up our next target as ‘url’:

```
>> C:\Users\Rishita\Desktop\Store_files\rss_xml_parser.js
        [24] function get_xml_file (url) {
        [27] // Precondition: must have a URL:
        [28] //alert(url);
        [29] if (url == "") return;
        [31] httpreq.open("GET", url, true);
```

Proceeding, we focus on the object ‘httpreq’ now:

```
>> C:\Users\Rishita\Desktop\Store_files\rss_xml_parser.js
        [25] var httpreq = getHTTPObject();
        [31] httpreq.open("GET", url, true);
        [33] httpreq.onreadystatechange = function () {
        [34] if (httpreq.readyState == 4) {
        [37] //alert(httpreq.responseText)
        [38] xmlfile = httpreq.responseXML;
        [46] httpreq.send (null);
```

Now the variable ‘xmlfile’ seems interesting to check in this function:

```
>> C:\Users\Rishita\Desktop\Store_files\rss_xml_parser.js
        [38] xmlfile = httpreq.responseXML;
        [39] //alert(xmlfile);
        [40] processRSS ("rss_place", xmlfile);
```

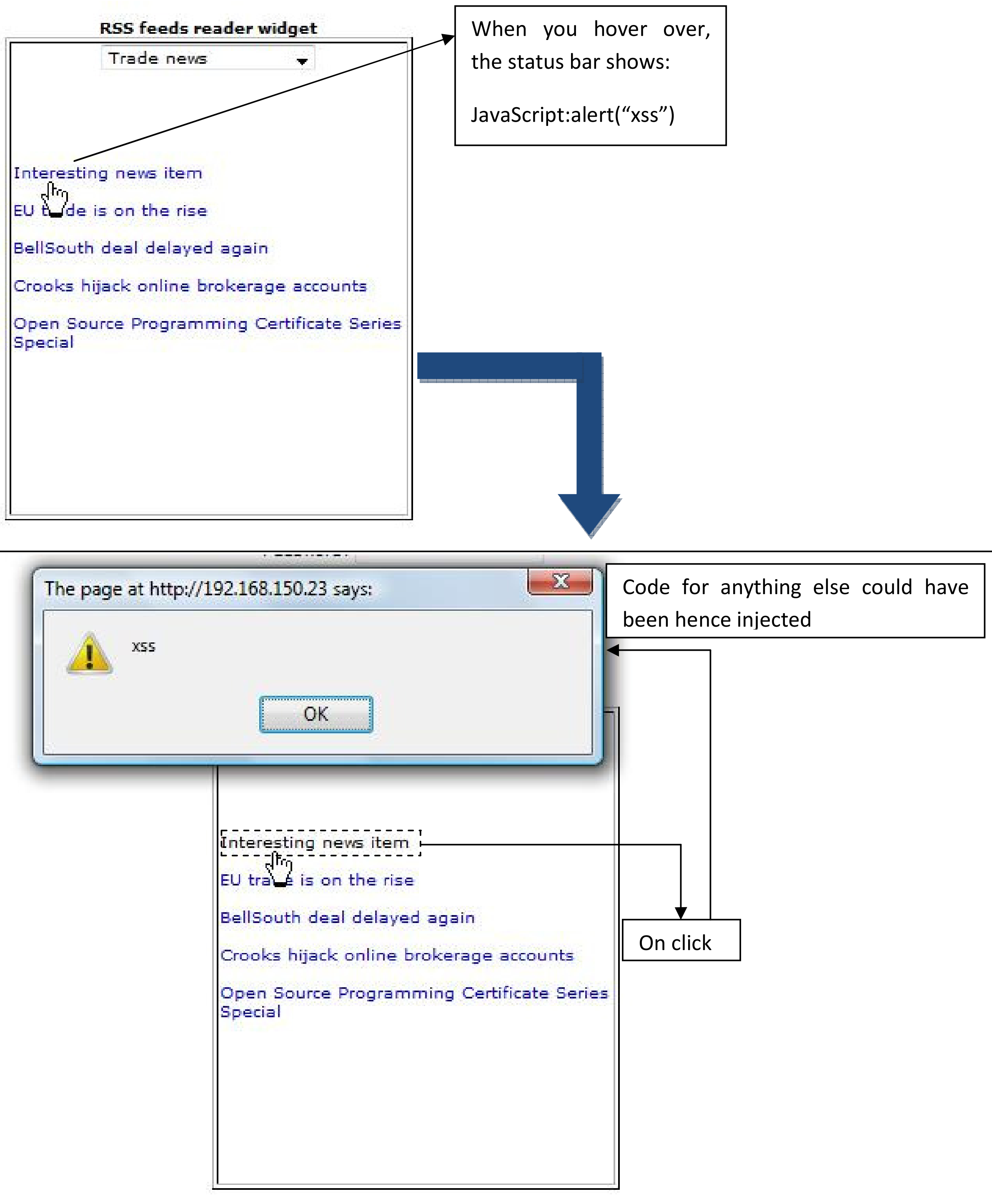
In this result that has come out, our interesting target seems ‘processRSS’:

```
>> C:\Users\Rishita\Desktop\Store_files\rss_xml_parser.js
        [5] function processRSS (divname, response) {
        [40] processRSS ("rss_place", xmlfile);
```

We move further to trace ‘response’ now:

```
>> C:\Users\Rishita\Desktop\Store_files\rss_xml_parser.js
        [1] var last_xml_response = "";
        [5] function processRSS (divname, response) {
        [7] var doc = response.documentElement;
        [37] //alert(httpreq.responseText)
        [38] xmlfile = httpreq.responseXML;
```

There we are! Without any validation on the requested url and data, the response has been obtained. This could be risky, which is demonstrated by the following screenshots in the sequence of the events (next page):




## Dynamic Code Debugging

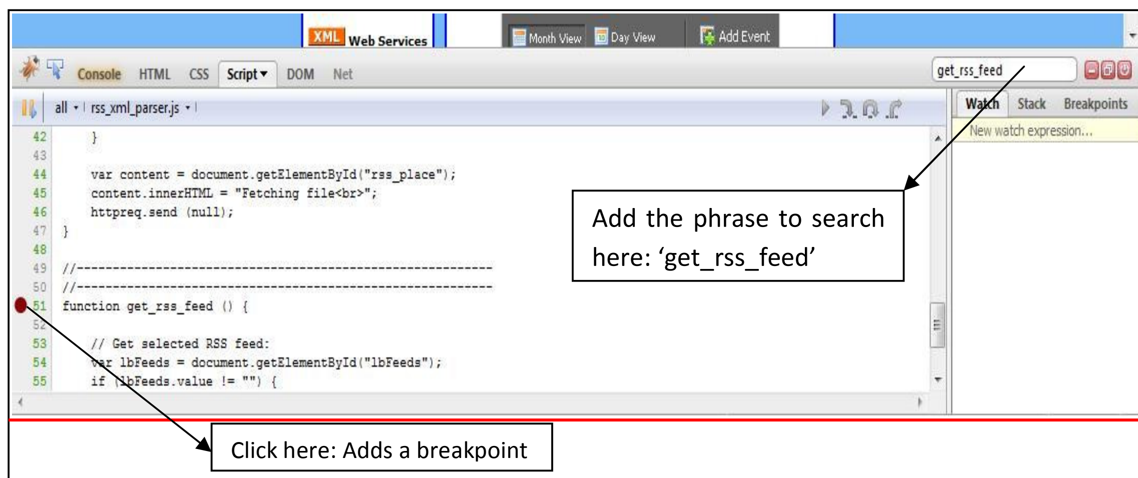
As far as this approach goes, the looking up of entire code will be avoided. Instead we will be on the lookout for areas in the rendered page which are popular for such vulnerabilities or look incriminating anyway. Hereafter we will try the XSS injection and analyse the working of the internal codes by inspecting them via a JavaScript debugger.

The tool that would be convenient to use for this purpose as a debugger for JavaScript code would be a Firefox extension called Firebug (http://getfirebug.com/).

In this, we can identify the area on the page then use the extension's capacity to map the area to the corresponding HTML block and thereafter set a breakpoint on the function being called. Once that is done, working on the page will be followed up in the Firebug panel on the side by correspondingly highlighted function call-returns. By setting successive breakpoints in this manner and analysing the XHR requests being made and the validation used if any, along with runtime data supplying to HTML elements such as forms, a dynamic analysis of the page with respect to XSS security can be done.

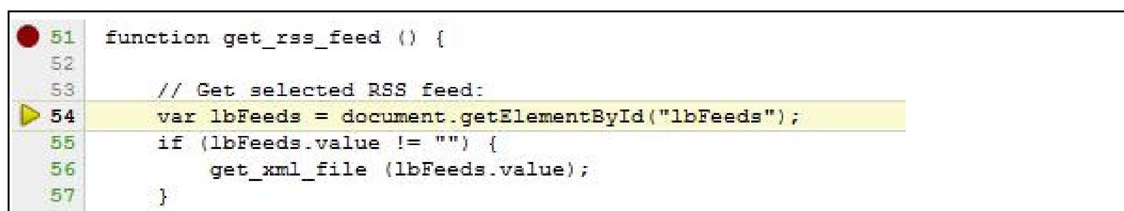
Once again, the detailed demonstration of a sample case will be accompanied by screenshots for a particular case in the Dynamic Approach Detailing below.

In the firebug panel shown, we input the function-name: 'get_rss_feed' and the 'Script' area in the left frame will get us to the function header. On clicking the area marked 'Click here' we can add a breakpoint there.




Now, as and when we click on the actual page anywhere the function is traced into by using the 'play, step into, step over' functions in the right hand top corner of the left frame in the firebug panel.

Now as we select 'Trade News' from the dropdown list shown on the page ('Trade News' is where the XSS exploit has been planted), as we saw in the HTML code earlier due to the event 'onchange' the function get_rss_feed is called and our breakpoint is encountered. Now on pressing 'Step into' the trace begins. The screenshot shown below is that when on the first pressing of 'Step Into' the line 54 is highlighted to show current line of operation.



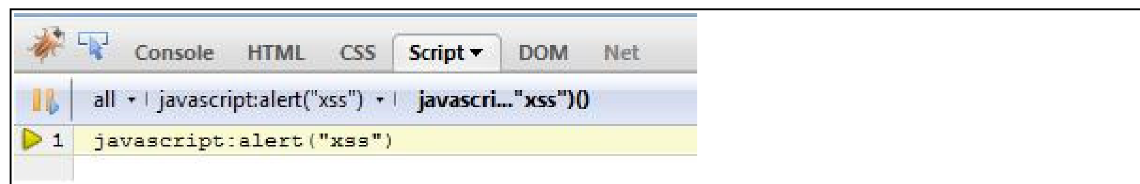
We follow the trace to line 56 and then the call to get_xml_file is made and shown as follows in the trace:

```
24  function get_xml_file (url) {
25      var httpreq = getHTTPObject();
26
27      // Precondition: must have a URL:
28      //alert(url);
29      if (url == "") return;
30
31      httpreq.open("GET", url, true);
32
33      httpreq.onreadystatechange = function () {
34      if (httpreq.readyState == 4) {
```

On tracing further we reach the following stage where line 31 is the operative line:

```
31      httpreq.open("GET", url, true);
32
33      httpreq.onreadystatechange = function () {
34      if (httpreq.readyState == 4) {
35          var content = document.getElementById("rss_place");
36          content.innerHTML = "Parsing...<br>";
37        //alert(httpreq.responseText)
38          xmlfile = httpreq.responseXML;
39              //alert(xmlfile);
40              processRSS ("rss_place", xmlfile);
```

Now the page is loaded with the ‘Trade News’ widget too. Now on tracing the click on the ‘Interesting News Item’ the following appears:




And there we are once again! Starting from choosing the item to here, we have not validated against such code snippets and therefore, this will be executed in the same manner as shown in the final result in the ‘Static Code Analysis Section’.

## Conclusion

Methodologies and approach described in this paper can help in discovering vulnerabilities in JavaScript driven Web 2.0 applications. The popular lookout areas apart from DOM based XSS are:

- Password comparisons in .js
- Business Logic Leaks in the client side: example: price
- Evals() used without appropriate validations
- All XHR requests without filters for ‘script’, ‘JavaScript’ tags and commands
- Injections by Unicode or escape sequences
- Direct reflection of the input from user forms
- Cookies not limited or localised to IPs and thus permitted cookie leakages
- Server side filters to avoid stored XSS
- Unescaped usage of < or > can permit tags more easily too
- Emails unescaped could increase the spamming disadvantages