

Hacking Web 2.0 Applications with Firefox

Shreeraj Shah 2006-10-11

Introduction

AJAX and interactive web services form the backbone of “web 2.0” applications. This technological transformation brings about new challenges for security professionals.

This article looks at some of the methods, tools and tricks to dissect web 2.0 applications (including Ajax) and discover security holes using Firefox and its plugins. The key learning objectives of this article are to understand the:

- web 2.0 application architecture and its security concerns.
- hacking challenges such as discovering hidden calls, crawling issues, and Ajax side logic discovery.
- discovery of XHR calls with the *Firebug* tool.
- simulation of browser event automation with the *Chickenfoot* plugin.
- debugging of applications from a security standpoint, using the Firebug debugger.
- methodical approach to vulnerability detection.

Web 2.0 application overview

The newly coined term “web 2.0” refers to the next generation of web applications that have logically evolved with the adoption of new technological vectors. XML-driven web services that are running on SOAP, XML-RPC and REST are empowering server-side components. New applications offer powerful end-user interfaces by utilizing Ajax and rich internet application (Flash) components.

This technological shift has an impact on the overall architecture of web applications and the communication mechanism between client and server. At the same time, this shift has opened up new security concerns [ref 1] and challenges.

New worms such as *Yamanner*, *Samy* and *Spaceflash* are exploiting “client-side” AJAX frameworks, providing new avenues of attack and compromising confidential information.

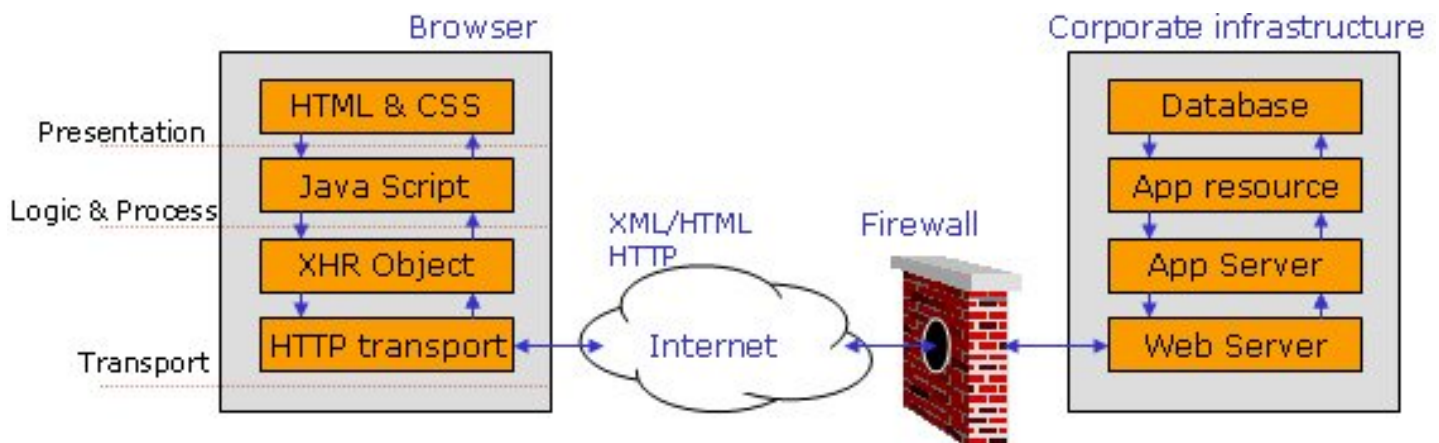


Figure 1. Web 2.0 architecture layout.

As shown in Figure 1, the browser processes on the left can be divided into the following layers:

- *Presentation layer* - HTML/CSS provides the overall appearance to the application in the browser window.
- *Logic & Process* - JavaScript running in the browser empowers applications to execute business and communication logic. AJAX-driven components reside in this layer.
- *Transport* - XMLHttpRequest (XHR) [ref 2]. This object empowers asynchronous communication capabilities and XML exchange mechanism between client and server over HTTP(S).

The server-side components on the right of Figure 1 that typically reside in the corporate infrastructure behind a firewall may include deployed web services along with traditional web application resources. An Ajax resource running on the browser can directly talk to XML-based web services and exchange information without *refreshing* the page. This entire communication is hidden from the end-user, in other words the end-user would not “feel” any *redirects*. The use of a “Refresh” and “Redirects” were an integral part of the first generation of web application logic. In the web 2.0 framework they are reduced substantially by implementing Ajax.

Web 2.0 assessment challenges

In this asynchronous framework, the application does not have many “Refreshes” and “Redirects”. As a result, many interesting server-side resources that can be exploited by an attacker are hidden. The following are three important challenges for security people trying to understand web 2.0 applications:

1. *Discovering hidden calls* - It is imperative that one identify XHR-driven calls generated by the loaded page in the browser. It uses JavaScript over HTTP(S) to make these calls to the backend servers.
2. *Crawling challenges* - Traditional crawler applications fail on two key fronts: one, to replicate browser behavior and two, to identify key server-side resources in the process. If a resource is accessed by an XHR object via JavaScript, then it is more than likely that the crawling application may not pick it up at all.
3. *Logic discovery* - Web applications today are loaded with JavaScript and it is difficult to isolate the logic for a particular event. Each HTML page may load three or four JavaScript resources from the server. Each of these files may have many functions, but the event may be using only a very small part of all these files for its execution logic.

We need to investigate and identify the methodology and tools to overcome these hurdles during a web application assessment. For the purpose of this article, we will use Firefox as our browser and try to leverage some of its plugins to combat the above challenges.

Discovering hidden calls

Web 2.0 applications may load a single page from the server but may make several XHR object calls when constructing the final page. These calls may pull content or JavaScript from the

server asynchronously. In such a scenario, the challenge is to determine all XHR calls and resources pulled from the server. This is information that could help in identifying all possible resources and associated vulnerabilities. Let's start with a simple example.

Suppose we can get today's business news by visiting a simple news portal located at:

`http://example.com/news.aspx`

The page in the browser would resemble the screenshot illustrated below in Figure 2.



Figure 2. A simple news portal page.

Being a web 2.0 application, Ajax calls are made to the server using an XHR object. We can determine these calls by using a tool known as *Firebug* [ref 3]. Firebug is a plug-in to the Firefox browser and has the ability to identify XHR object calls.

Prior to browsing a page with the plugin, ensure the option to intercept XHR calls is selected, as shown in Figure 3.

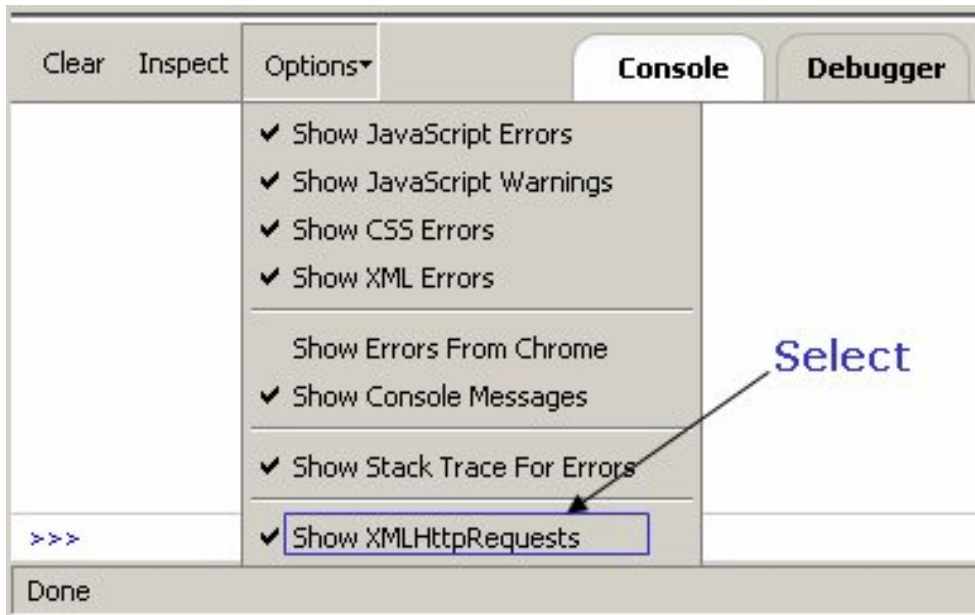


Figure 3. Setting Firebug to intercept XMLHttpRequest calls.

With the Firebug option to intercept XMLHttpRequest calls enabled, we browse the same page to discover all XHR object calls made by this particular page to the server. This exchange is shown in Figure 4.

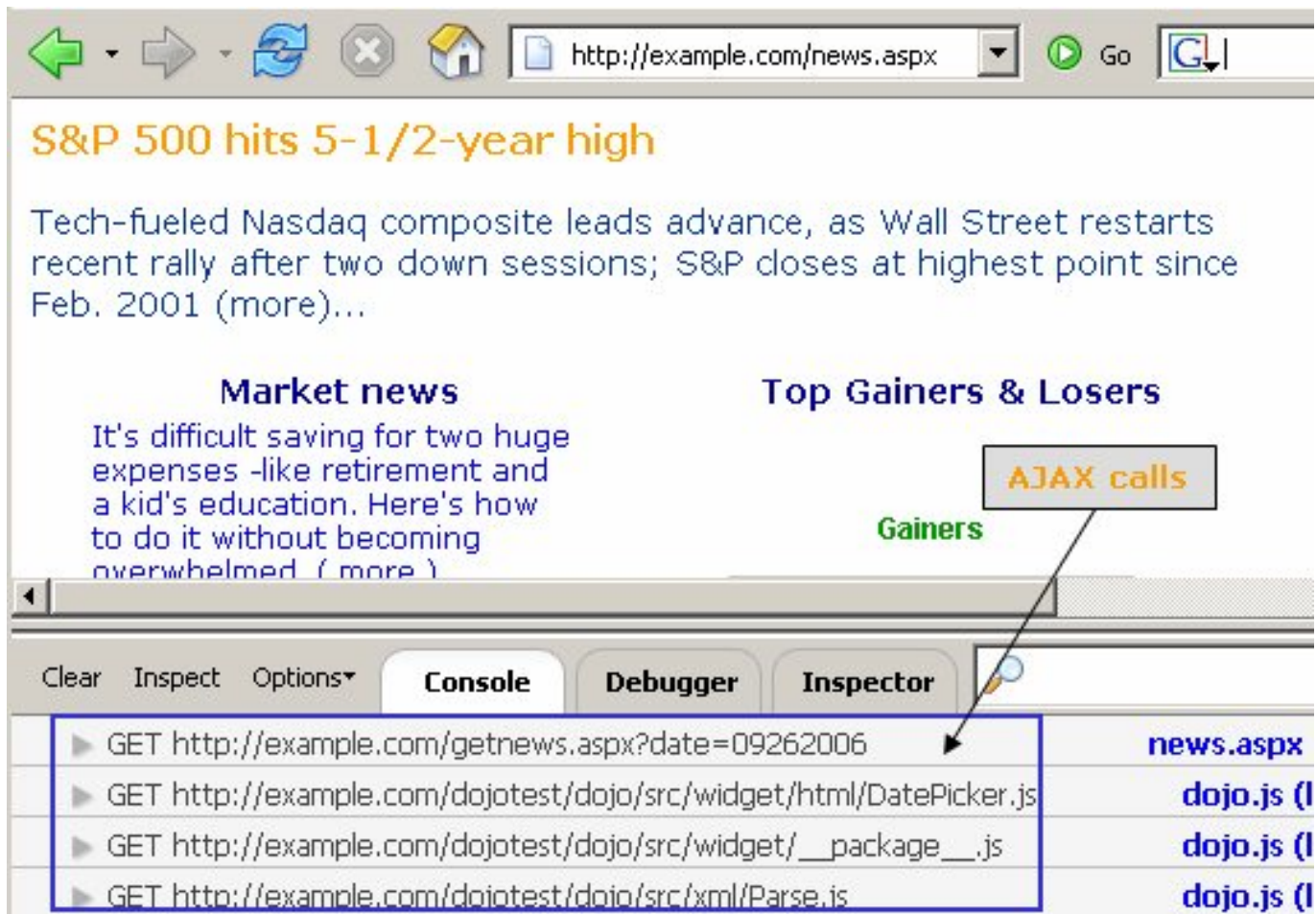


Figure 4. Capturing Ajax calls.

We can see several requests made by the browser using XHR. It has loaded the *dojo* AJAX

framework from the server while simultaneously making a call to a resource on the server to fetch news articles.

<http://example.com/getnews.aspx?date=09262006>

If we closely look at the code, we can see following function in JavaScript:

```
function getNews()
{
    var http;
    http = new XMLHttpRequest();
    http.open("GET", " getnews.aspx?date=09262006", true);
    http.onreadystatechange = function()
    {
        if (http.readyState == 4) {
            var response = http.responseText;
            document.getElementById('result').innerHTML = response;
        }
    }
    http.send(null);
}
```

The preceding code makes an asynchronous call to the backend web server and asks for the resource `getnews.aspx?date=09262006`. The content of this page is placed at the 'result' id location in the resulting HTML page. This is clearly an Ajax call using the XHR object.

By analyzing the application in this format, we can identify vulnerable internal URLs, *querystrings* and POST requests as well. For example, again using the above case, the parameter "date" is vulnerable to an SQL injection attack.

continued

[ref 1] Ajax security, <http://www.securityfocus.com/infocus/1868>

[ref 2] XHR Object specification, <http://www.w3.org/TR/XMLHttpRequest/>

[ref 3] Firebug download, <https://addons.mozilla.org/firefox/1843/>; Firebug usage, <http://www.joehewitt.com/software/firebug/docs.php>

Crawling challenges and browser simulation

An important reconnaissance tool when performing web application assessment is a web crawler. A web crawler crawls every single page and collects all HREFs (links). But what if these HREFs point to a JavaScript function that makes Ajax calls using the XHR object? The web crawler may miss this information altogether.

In many cases it becomes very difficult to simulate this environment. For example, here is a set of simple links:

```
<a href="#" onclick="getMe(); return false;">go1</a><br>
<a href="/hi.html">go2</a><br>
<a href="#" onclick="getMe(); return false;">go3</a><br>
```

The "go1" link when clicked will execute the `getMe()` function. The code for `getMe()` function is as shown below. Note that this function may be implemented in a completely separate file.

```
function getMe()
{
    var http;
    http = new XMLHttpRequest();
    http.open("GET", "hi.html", true);
    http.onreadystatechange = function()
    {
        if (http.readyState == 4) {
            var response = http.responseText;
            document.getElementById('result').innerHTML = response;
        }
    }
    http.send(null);
}
```

The preceding code makes a simple Ajax call to the `hi.html` resource on the server.

Is it possible to simulate this click using automation? Yes! Here is one approach using the Firefox plug-in *Chickenfoot* [ref 4] that provides JavaScript-based APIs and extends the programmable interface to the browser.

By using the *Chickenfoot* plugin, you can write simple JavaScript to automate browser behavior. With this methodology, simple tasks such as crawling web pages can be automated with ease. For example, the following simple script will "click" all anchors with `onClick` events. The advantage of this plug-in over traditional web crawlers is distinct: each of these `onClick` events makes backend XHR-based AJAX calls which may be missed by crawlers because crawlers try to parse JavaScript and collect possible links but cannot replace actual `onClick` events.

```
l=find('link')
for(i=0;i<l.count;i++){
a = document.links[i];
test = a.onclick;
if(!(test== null)){
```

```

var e = document.createEvent('MouseEvents');
e.initMouseEvent('click',true,true,document.defaultView,1,0,0,0,
                0,false,false,false,false,0,null);

a.dispatchEvent(e);
}

```

You can load this script in the *Chickenfoot* console and run it as shown in Figure 5.

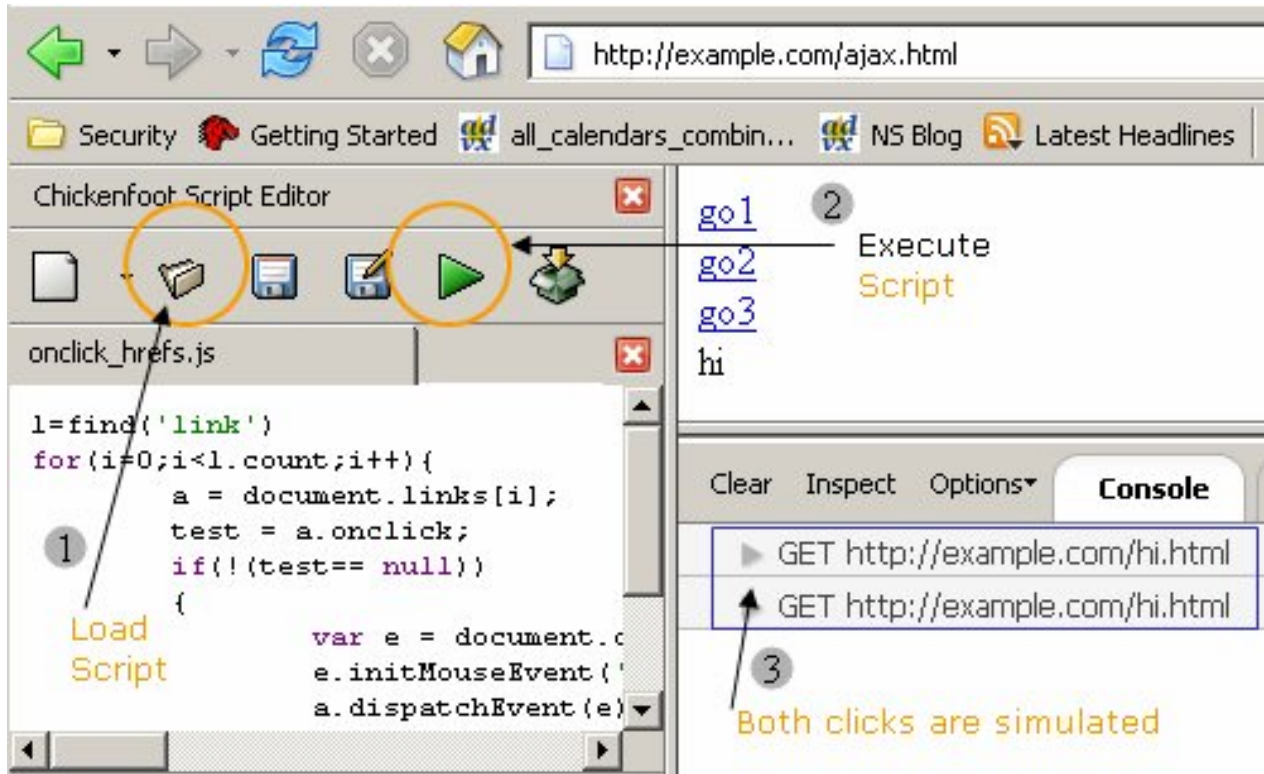


Figure 5. Simulating onClick AJAX call with chickenfoot.

This way, one can create JavaScript and assess AJAX-based applications from within the Firefox browser. There are several API calls [ref 5] that can be used in the chickenfoot plugin. A useful one is the “fetch” command to build a crawling utility.

Logic discovery & dissecting applications

To dissect client-side Ajax-based applications, one needs to go through each of the events very carefully in order to determine process logic. One way of determining the entire logic is to walk through each line of code. Often, each of these event calls process just a few functions from specific files only. Hence, one needs to use a technique to step through the relevant code that gets executed in a browser.

There are a few powerful debuggers for JavaScript that can be used to achieve the above objective. *Firebug* is one of them. Another one is *venkman* [ref 6]. We shall use Firebug again in our example.

Let’s take a simple example of a login process. The `login.html` page accepts a username and

password from the end-user, as shown in Figure 6. Use the “inspect” feature of Firebug to determine the property of the form.

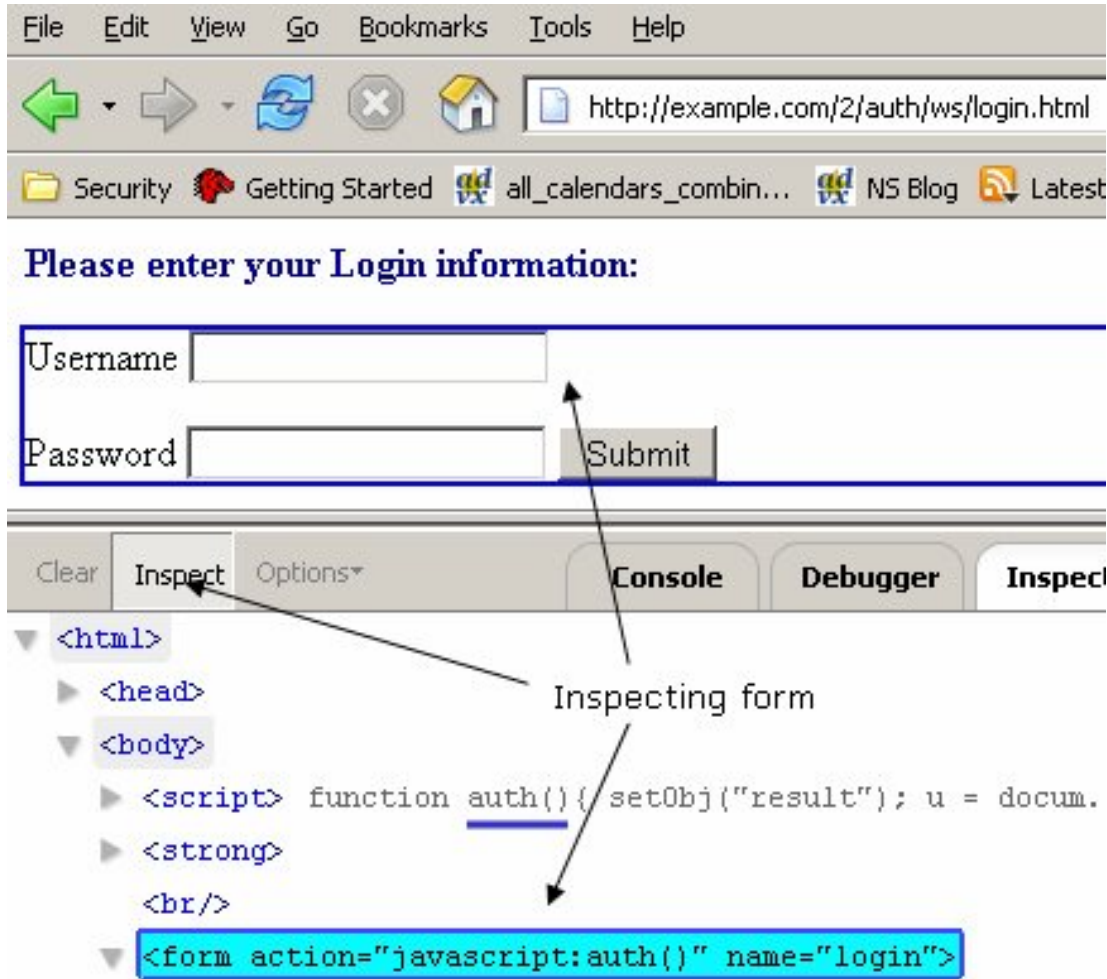


Figure 6. Form property inspection with Firebug.

After inspecting the form property, it is clear that a call is made to the “auth” function. We can now go to the debugger feature of Firebug as illustrated in Figure 7 and isolate internal logic for a particular event.

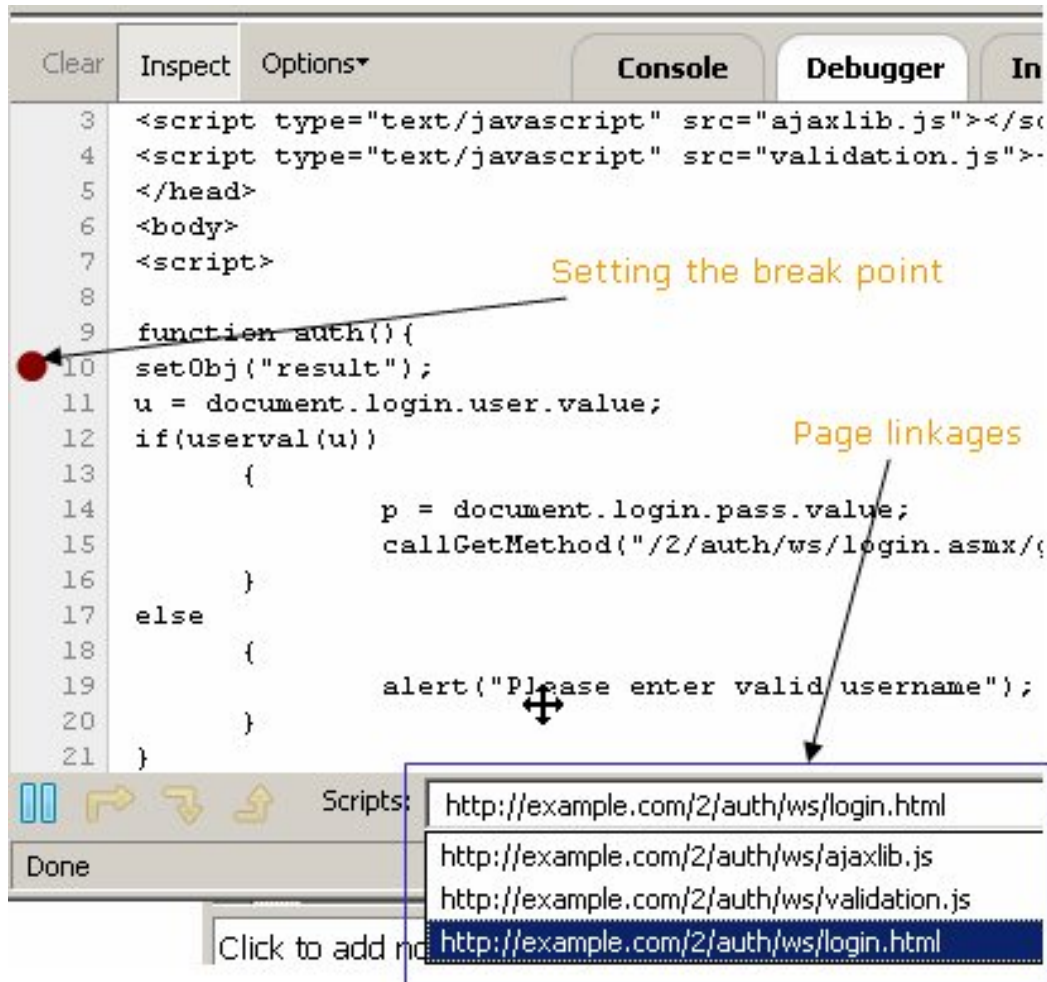


Figure 7. Debugging with Firebug.

All JavaScript dependencies of this particular page can be viewed. Calls are made to the *ajaxlib.js* and *validation.js* scripts. These two scripts must have several functions. It can be deduced that the login process utilizes some of these functions. We can use a "breakpoint" to step through the entire application. Once a breakpoint is set, we can input credential information, click the "Submit" button and control the execution process. In our example, we have set a breakpoint in the "auth" function as shown in Figure 8.

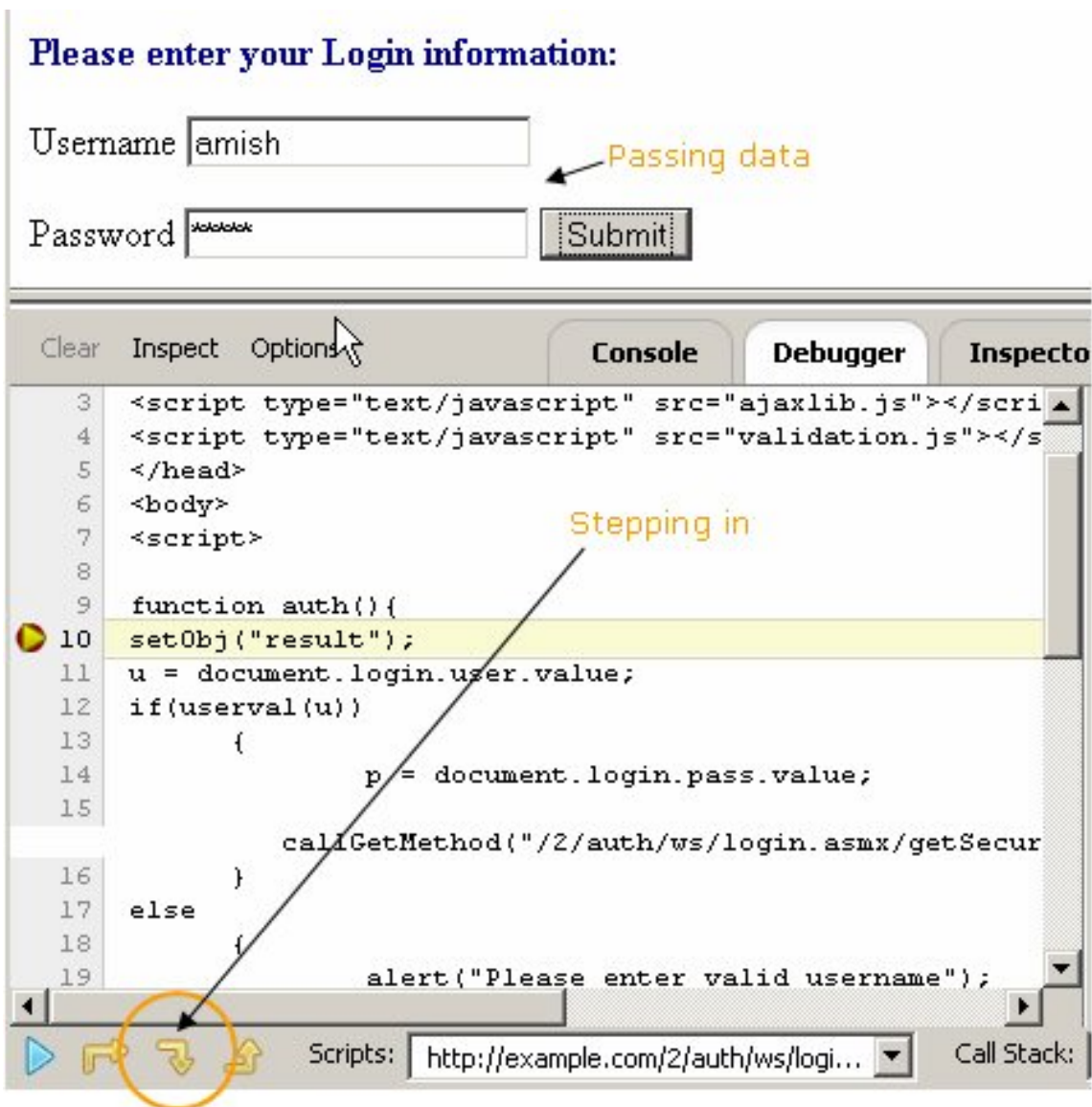


Figure 8. Setting a breakpoint and controlling execution process.

We now step through the debugging process by clicking the "step in" button, which was highlighted in Figure 8. JavaScript execution moves to another function, *userval*, residing in the file *validation.js* as shown in Figure 9.

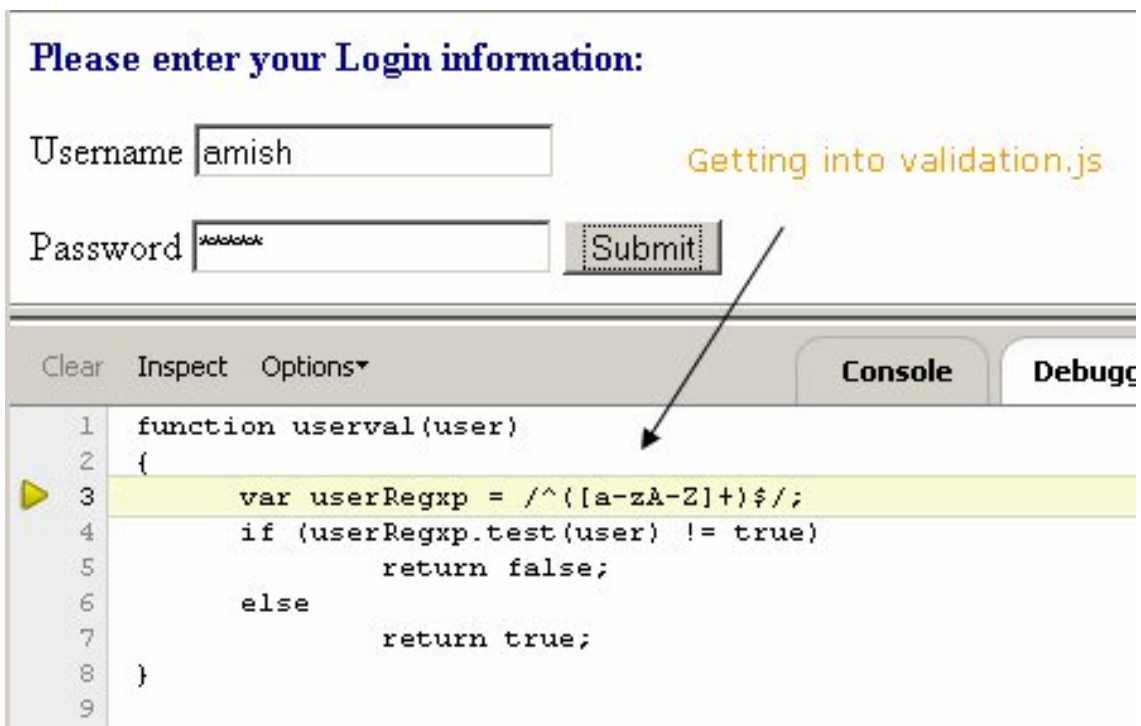


Figure 9. Moving to validation.js script page.

The preceding screenshot shows the regular expression pattern used to validate the username field. Once validation is done execution moves to another function `callGetMethod` as shown in Figure 10.

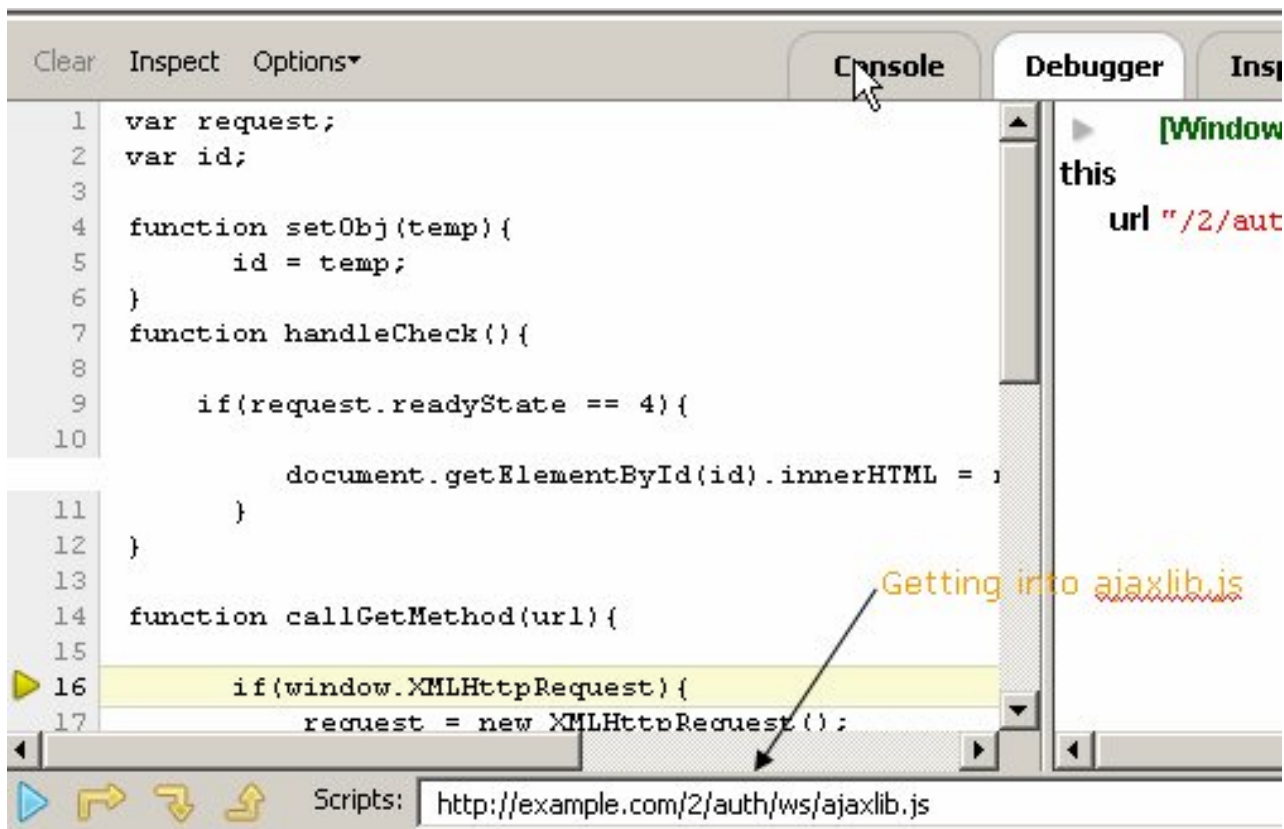


Figure 10. Making an Ajax call.

Finally, at the end of the execution sequence, we can observe the call to backend web services as being made by the XHR object. This is shown in Figure 11.

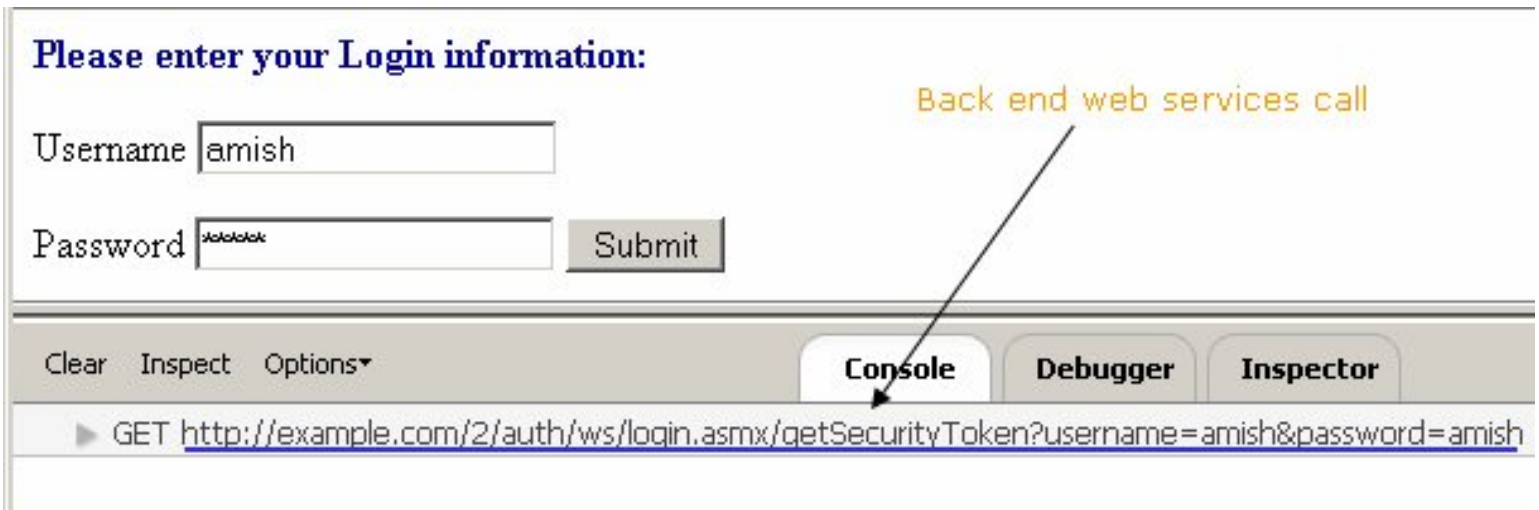


Figure 11. Web services call on the Firebug console.

Here we have identified the resource location for the backend web services:

`http://example.com/2/auth/ws/login.asmx/getSecurityToken?username=amish&password=amish`

The preceding resource is clearly some web services running under the .NET framework. This entire dissection process has thrown up an interesting detail: we've found a user validation routine that can be bypassed very easily. It is a potential security threat to the web application.

Taking our assessment further, we can now access the web service and its endpoints by using a WSDL file and directly bruteforce the service. We can launch several different injection attacks - SQL or XPATH - with tools such as wsChess [ref 7].

In this particular case, the application is vulnerable to an XPATH injection. The methodology for web services assessment overall is different and is outside the scope of this article. However this walkthrough technique helps identify several client-side attacks such as XSS, DOM manipulation attacks, client-side security control bypassing, malicious Ajax code execution, and so on.

Conclusion

Service-oriented architecture (SOA), Ajax, Rich Internet Applications (RIA) and web services are critical components to next generation web applications. To keep pace with these technologies and combat next-generation application security challenges, one needs to design and develop different methodologies and tools. One of the efficient methodologies of assessing applications is by effectively using a browser.

In this article we have seen three techniques to assess web 2.0 applications. By using these methodologies it is possible to identify and isolate several Ajax-related vulnerabilities. Browser automation scripting can assist us in web asset profiling and discovery, that in turn can help in identifying vulnerable server-side resources.

Next generation applications use JavaScript extensively. Smooth debugging tools are our knights in shining armor. The overall techniques covered in this article is a good starting point for web 2.0 assessments using Firefox.

References

[ref 1] Ajax security,

<http://www.securityfocus.com/infocus/1868>

[ref 2] XHR Object specification, <http://www.w3.org/TR/XMLHttpRequest/>

[ref 3] Firebug download, <https://addons.mozilla.org/firefox/1843/>; Firebug usage, <http://www.joehewitt.com/software/firebug/docs.php>

[ref 4] Chickenfoot quick start, <http://groups.csail.mit.edu/uid/chickenfoot/quickstart.html>

[ref 5] Chickenfoot API reference - <http://groups.csail.mit.edu/uid/chickenfoot/api.html>

[ref 6] Venkman walkthrough, <http://www.mozilla.org/projects/venkman/venkman-walkthrough.html>

[ref 7] wsChess, <http://net-square.com/wsches>

About the author

Shreeraj Shah, BE, MSCS, MBA, is the founder of Net Square and leads Net Square's consulting, training and R&D activities. He previously worked with Foundstone, Chase Manhattan Bank and IBM. He is also the author of *Hacking Web Services* (Thomson) and co-author of *Web Hacking: Attacks and Defense* (Addison-Wesley). In addition, he has published several advisories, tools, and whitepapers, and has presented at numerous conferences including RSA, AusCERT, InfosecWorld (Misti), HackInTheBox, Blackhat, OSCON, Bellua, Syscan, etc. You can read his blog at <http://shreeraj.blogspot.com/>.

Reprints or translations

Reprint or translation requests require [prior approval](#) from SecurityFocus.

© 2006 SecurityFocus

Comments?

Public comments for Infocus articles, below, require technical merit to be published. General comments, article suggestions and feedback are encouraged but should be sent to the [editorial team](#) instead.

[Privacy Statement](#)

Copyright 2006, SecurityFocus