Vulnerability Scanning Web 2.0 Client-Side Components

Shreeraj Shah 2006-11-27

Introduction

Web 2.0 applications are a combination of several technologies such as Asynchronous JavaScript and XML (AJAX), Flash, JavaScript Object Notation (JSON), Simple Object Access Protocol (SOAP), Representational State Transfer (REST). All these technologies, along with cross-domain information access, contribute to the complexity of the application. We are seeing a shift towards empowerment of an end-user's browser by loading libraries.

All these changes mean new scanning challenges for tools and professionals. The key learning objectives of this article are to understand the following concepts and techniques:

- Scanning complexity and challenges in new generation Web applications
- Web 2.0 client-side scanning objectives and methodology
- Web 2.0 vulnerability detection (XSS in RSS feeds)
- Cross-domain injection with JSON
- Countermeasures and defense through browser-side filtering

Web 2.0 scanning complexities

The next generation Web 2.0 applications are very complex in nature and throw up new scanning challenges. The complexities can be attributed to the following factors:

- *Rich client interface* - AJAX and Flash provide rich interfaces to applications with complex JavaScripts and Actionscripts, making it difficult to identify application logic and critical resources buried in these scripts.
- *Information sources* - Applications are consuming information from various sources and building up mashups [ref 1] within sites. An application aggregates RSS feeds or blogs from different locations and builds a large repository of information at a single location.
- *Data structures* - Exchange of data between applications is done using XML, JSON [ref 2], Java script arrays and proprietary structures.
- *Protocols* - Aside from the simple HTTP GET and POST, applications can choose from an array of different protocols such as SOAP, REST and XML-RPC.

Our target application may be accessing RSS feeds from multiple sites, exchanging information with blogs using JSON, and communicating with a stock exchange portal's Web service over SOAP. All these services are bundled in the form of Rich Internet Applications (RIA) using AJAX and/or Flash.

Web 2.0 application scanning challenges

Application scanning challenges can be divided into two parts:

1. ***Scanning server-side application components*** - One of the biggest challenges when scanning Web 2.0 applications is to identify buried resources on the server. When scanning traditional applications, a crawler can be run that would look for the string "href" in order to identify and profile Web application assets.

   In the case of Web 2.0 applications, however, one needs to identify backend Web services, third-party mashup, backend proxies, etc. The author has addressed some of these challenges in a previous article [ref 3].
2. ***Scanning client-side application components*** - A Web 2.0 application can loads several JavaScripts, Flash components, and widgets in the browser. These scripts and components utilize the *XMLHTTPRequest* object to communicate with the backend Web server. It is also possible to access cross-domain information from within the browser itself. Cross-site scripting (XSS) attacks [ref 4] are potential threats to the application user. The Web 2.0 framework uses various client-side scripts and consumes information from "untrusted third-party sources." AJAX and JSON technologies, cross-domain access and dynamic DOM manipulation techniques are adding new dimensions to old XSS attacks [ref 5]. Client-side component scanning and vulnerability detection in Web 2.0 are new challenges coming up on the horizon. The scope of this article is restricted to this scanning category.

Client-side scanning objectives

To understand these scanning objectives clearly, let us take a sample scenario as illustrated in Figure 1.0. We have a Web application running on *example.com*. Clients access this application via a Web browser.

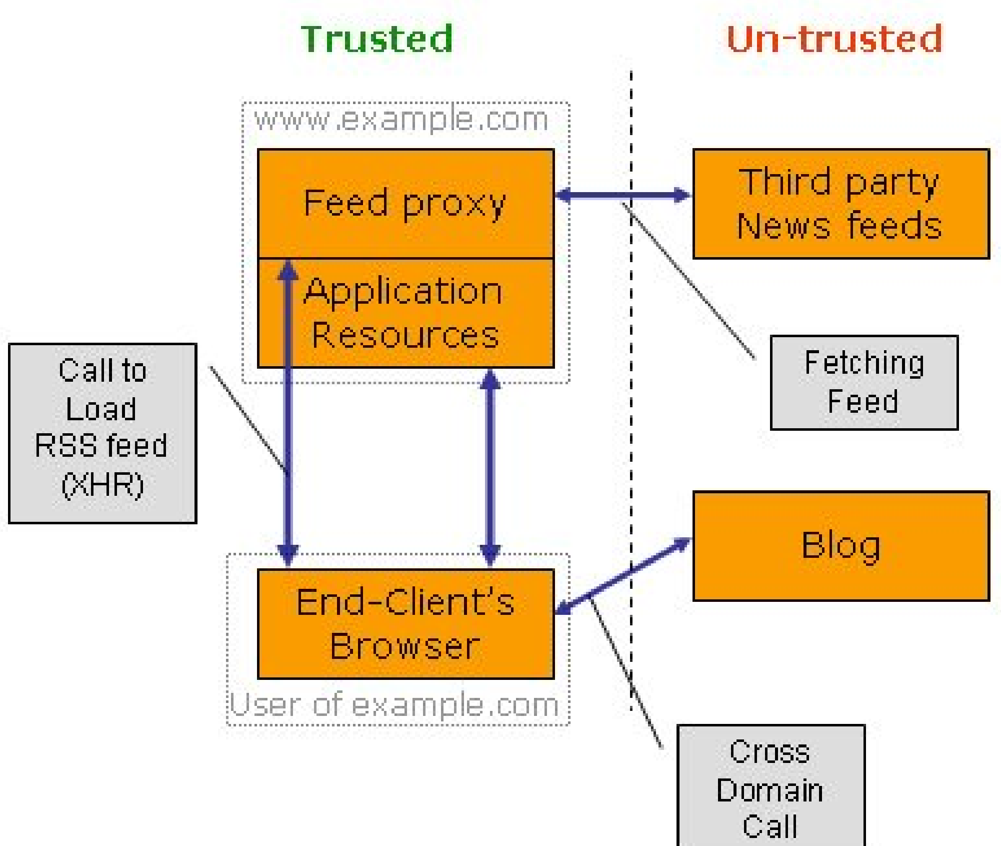

Figure 1. Web 2.0 target application layout.

This web application can be divided into the following sections with regard to their usage and logic.

Application resources- these resources are deployed by *example.com* and they can be of any type: HTML, ASP/JSP, Web services. All these resources are in a fully trusted domain and are owned by *example.com*.

Feed proxy - The *XMLHTTPRequest* object cannot make direct backend calls to cross-domains. To circumvent this restriction a proxy is set up by *example.com*that can give access to third-party RSS feeds, for example, a daily news feed. Hence, users of *example.com* can set up any feed on the Internet for daily use.

Blog access - End-users use the same application loaded by *example.com* to access some of the blogs on the Internet. This is possible because *example.co*m loads certain scripts on the client's browser that allow users to access cross-domain blogs.

Here are four critical scanning objectives to determine client-side vulnerabilities.

1. *Technology and library fingerprinting* - Web 2.0 applications can be created by many AJAX and Flash libraries. These libraries get loaded in the browser and are used by the application as and when needed. It is important to fingerprint these libraries and map them to publicly known vulnerabilities.
2. *Third-party untrusted information points* - In Figure 1, we have divided the Web

application layout into "trusted" and "untrusted" areas. Information originating from untrusted sources needs thorough scrutiny prior to loading it in the browser. In our example this information flows via an application server proxy in the case of news feeds, and directly into the DOM in the case of blogs.

3. *DOM access points* - The browser runs everything in its DOM context. Loaded JavaScripts manipulate the DOM. If malicious information is passed to any one of these access points the browser can be at risk. DOM access points are therefore essential bits of information.
4. *Functions and variable traces for vulnerability detection* - Once DOM access points and third-party information has been identified, it is important to understand execution logic and corresponding traces in the browser in order to expose threats and vulnerabilities.

Scanning client-side applications [news feeds]

In this section we shall adopt a manual approach to the scanning process. This methodology can be automated to some extent but given the complexity of the application it may be difficult to scan for all possible combinations.

The target resource -*http://example.com/rss/news.aspx*

We get the following page, as shown below in Figure 2.

Figure 2. RSS feed application widget.

The above page serves various RSS feeds configured by the end-user. Now let's walk through the steps we require for scanning.

1. Scanning for technology and fingerprints

All possible JavaScripts consumed by the browser after loading the page can be grabbed from the HTML page itself by viewing the HTML source, as listed in Figure 3, or programmatically using regular expressions.

```
view-source: - Source of: http://example.com/rss/news.aspx - Mozilla Firefox
File  Edit  View  Help
-->
</style>
<head>
<title>News RSS feed</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-
<script type="text/javascript" src="dojo.js"></script>
<script language="javascript" src="rss_xml_parser.js"></script>
<script language="javascript" src="XMLHTTPReq.js"></script>
```

Figure 3. All JavaScripts for the application page.

If you have the Firefox plugin "Web Developer" [ref 6], you can view all scripts in a single page as shown below in Figure 4.

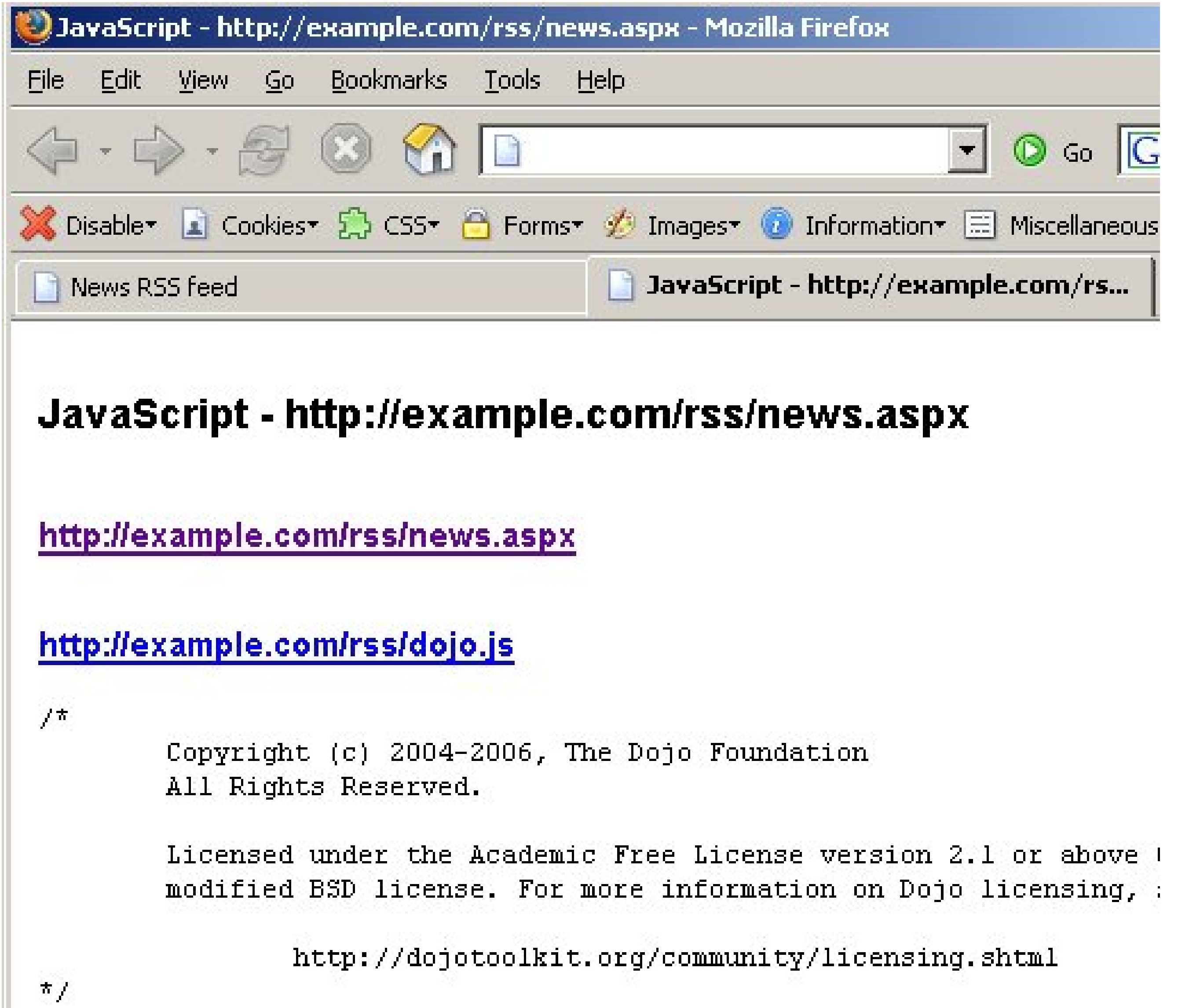

Figure 4. All JavaScripts along with source code.

The following information can be identified by scanning these JavaScripts:

- One of the AJAX development toolkits *dojo.js* [ref 7] is being used. File names provide vital clues when fingerprinting these technologies. We can scan the content further to determine the version in use. A similar technique can also be employed to fingerprint Microsoft's Atlas and many other technologies. This information helps in mapping known vulnerabilities to underlying architecture.
- Files containing functions that are consumed by an RSS feed application can be mapped within the browser. Here is a brief list:
  - The *rss_xml_parser.js* file contains functions such as `processRSS()` and `GetRSS()`. These functions fetch RSS feeds from the server and process them.
  - The *XMLHTTPReq.js*file contains `makeGET()` and `makePOST()` functions to process AJAX requests.
  - The *dojo.js* file contains several other functions.

All this enumerated information can be organized to obtain a better picture of the process.

2. Third party untrusted information points

We scan the HTML source for the page and locate the following code:

This code calls the function `GetRSS()`, which in turn, makes a request to the proxy to fetch untrusted RSS feeds from various servers.

Continued on page 2...

[ref 1] Brief on mashup (http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid))
[ref 2] JavaScript Object Notation (JSON) is a lightweight data-interchange format (http://www.json.org/)
[ref 3] Hacking Web 2.0 Applications with Firefox (http://www.securityfocus.com/infocus/1879)
[ref 4] XSS threat classification (http://www.webappsec.org/projects/threat/classes/cross-site_scripting.shtml)
[ref 5] DOM Based Cross Site Scripting or XSS of the Third Kind - By Amit Klein(http://www.webappsec.org/projects/articles/071105.shtml)
[ref 6] Web developer plugin (http://chrispederick.com/work/webdeveloper/)
[ref 7] Dojo toolkit (http://www.dojotoolkit.com/)

3. DOM access points

Having collected all JavaScripts, we can look for certain patterns where the DOM gets accessed. We look for *"document.*"* usage and then narrow down the search to two potential candidates:

document.getElementById(name).innerHTML - This function is used extensively by applications to change HTML layers dynamically.

document.write()- This function is also used to change HTML layers in the browser.

There may be a few other calls which transform DOM views in the browser. At this point, however, we shall focus on the two preceding functions. In the source, scroll down to the following function where "innerHTML" is used.

```
function processRSS (divname, response) {
    var html = "";
    var doc = response.documentElement;
    var items = doc.getElementsByTagName('item');
    for (var i=0; i < items.length; i++) {
        var title = items[i].getElementsByTagName('title')[0];
        var link = items[i].getElementsByTagName('link')[0];
                html += ""
                + title.firstChild.data
                + "
";
    }
    var target = document.getElementById(divname);
    target.innerHTML = html;
}
```

4. Functions and variable traces for vulnerability detection

We can organize all information gathered from the preceding three steps and apply debugging techniques [ref 8] to determine the entire flow of client-side logic. This is illustrated in Figure 5.

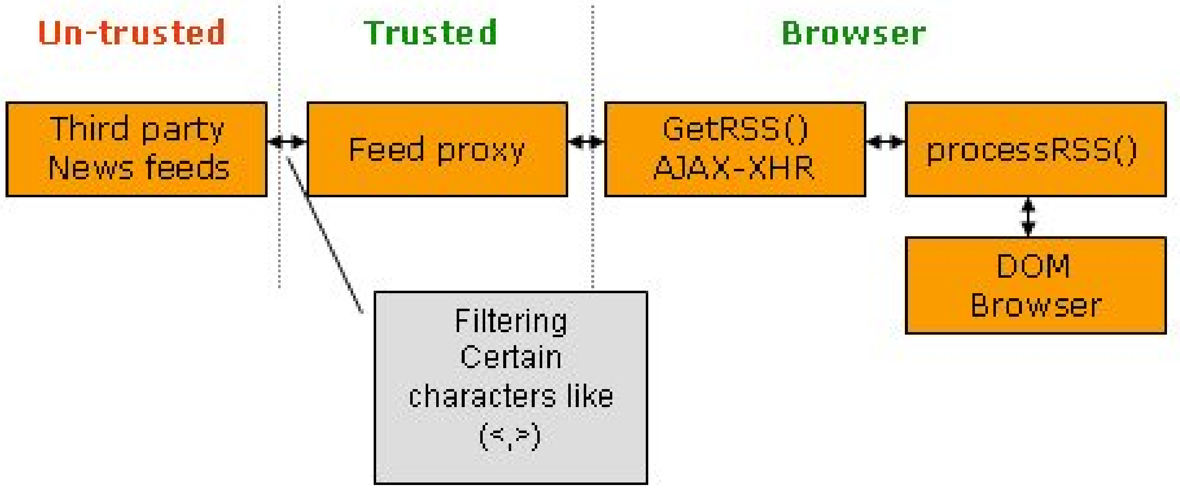

Figure 5. Execution logic and data flow for news feed function.

From this logic it is clear that the feed proxy is filtering out certain characters such as < and >, effectively making it impossible to inject JavaScript into the DOM. Again, since execution modifies "innerHTML" in preloaded DOM it is not possible to execute scripts either. Observe closely the following line from the loop that builds HTML dynamically:

```
html += ""
```

What if an untrusted RSS feed injects a malicious link? This link is not validated anywhere as is evident from the code. Here is an example of the RSS node:

```
javascript:alert("Simple XSS")

XYZ news
2005-11-16T16:00:00-08:00
```

Note that the "href" element of XML contains JavaScript. Hence, when an end-user clicks the link the script will run in the current DOM context. This process is illustrated below in Figure 6.

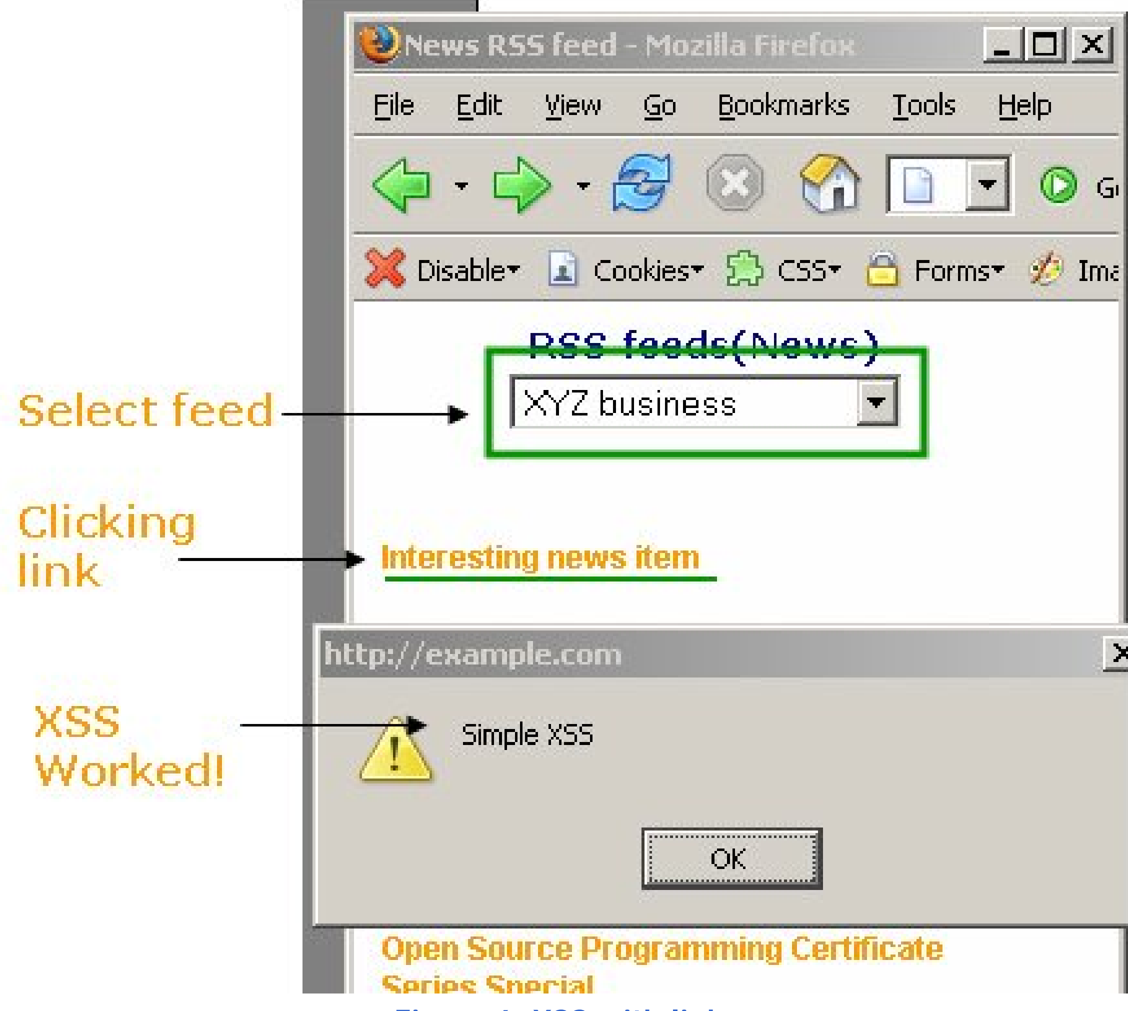

Figure 6. XSS with link.

Information presentation in the DOM from an untrusted source may put an end-user's session at risk. Web 2.0 applications provide information from different sources in a single browser page.

Let's take another example to understand this attack vector clearly.

Cross-domain JSON injection

JSON is a very lightweight structure for information exchange vis-à-vis XML that has a sizeable overhead. Many applications such as Google, Yahoo and others have extended their Web services with JSON callback. With this callback in place, cross-domain information can be captured and processed in specific functions.

Let's take an example. Assume that a web site running at *http://blog.example.org*, has extended their services using JSON callback to allow anyone access to information using JavaScript. A JSON structure with callback name will be provided.

Access the profile for `id=10` by pointing the browser to the location or by sending a GET to the following location:

Request:
http://blog.example.org/Getprofile.html?callback=profileCallback&id=10

Response:
```
profileCallback({"profile":[{"name":"John","email":"john@my.com"}]})
```

As you can see, we get JSON wrapped around `profileCallback`. With this callback the `profileCallback()` function gets executed with JSON output as its argument.

Similarly if we send `id=11` we get the following response back:
```
profileCallback({"profile":[{"name":"Smith","email":"smith@cool.com"}]})
```

Our target *example.com*has integrated this blog service in its application. If we scan their clien-side code and look for `document.write` we get the following code snippet in one of their pages: *showprofile.html*.

The code makes a call to "*blog.example.org*" which is a cross-domain and processes output in "profileCallback". By calling the page we get following output, shown below in Figure 7.

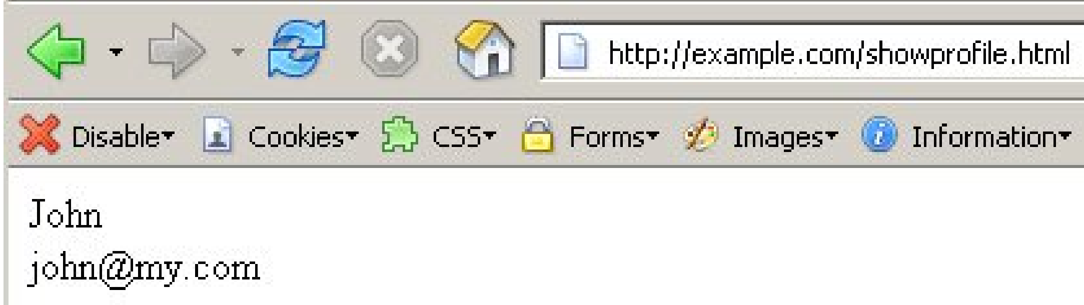

Figure 7. Simple JSON callback.

It is clear that this application running on *example.com* consumes third party untrusted JSON information into the application without any validation. This exposes an end-user's sensitive data such as session and cookie information.

This means that if one of the clients is trying to look up information for `id=101` that has an injected JavaScript into its email field, such as the one show below, will instead have a remote malicious JavaScript run on the machine.

```
profileCallback({"profile":[{"name":"Jack","email":""}]})
```

Simply put, the victim's machine can be compromised while the victim browses through *example.com*'s application as shown in Figure 8.

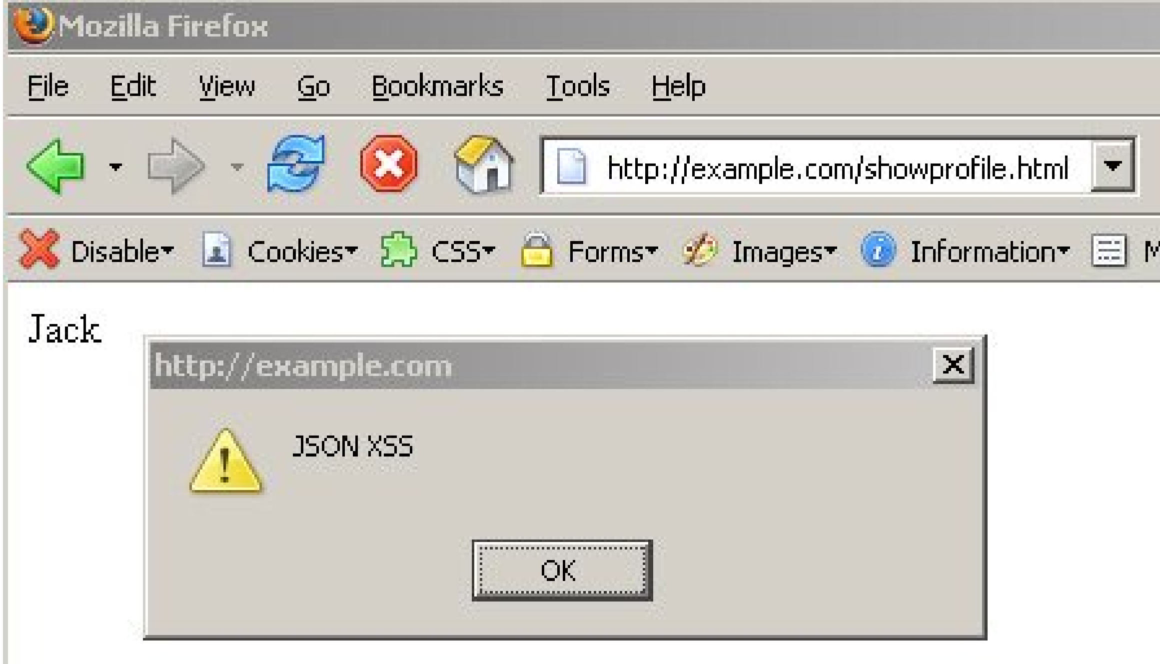

Figure 8. Possible XSS with JSON injection.

This way untrusted information processing in an AJAX virtual application sandbox can be exploited by poorly written client-side scripts or components. In this article we have covered two ways to pass on payloads to the end-user. A few other methods exist as well.

Countermeasures

To protect client-side browsers it is important to follow a re-worded maxim, "trust no third-party information." In the design phase of the application one needs to clearly define a virtual application sandbox and provide incoming information validation for all third-party sources appearing in the form of XML, RSS feeds, JSON, JavaScript arrays, etc. as shown below in Figure 9.

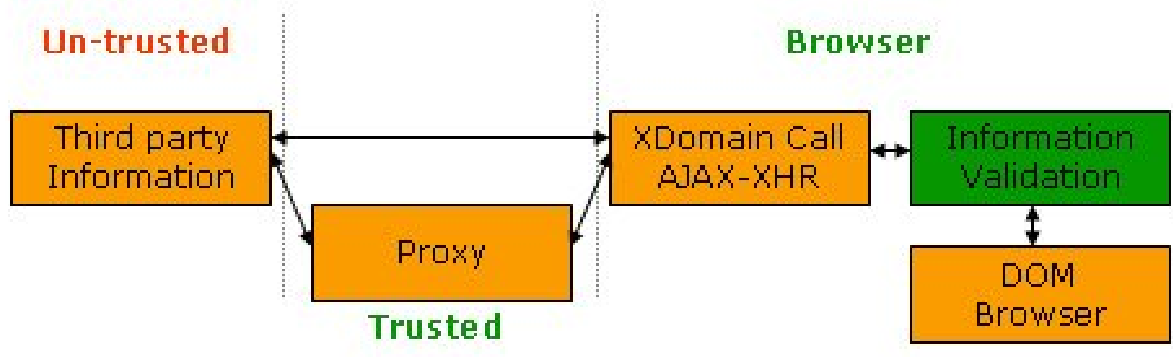

Figure 9. Virtual application sandbox to validate third-party information sources.

For example, prior to posting HREFs to the DOM, pass them to a simple function, such as the one shown in the code snippet below, to look for any JavaScript code injection.

```
function checkLink(link) {
   if(link.match(/javascript:|<|>/))
   {
      return false;
   } else {
      return true;
   }
}
```

It is important to filter all incoming traffic before it hits the DOM in a browser-side application sandbox - a final defense for the end-user.

Conclusion

In recent times several incidents [ref 9] were observed in which Web 2.0 applications were compromised at the client's end due to poorly written scripts. Future automated or manual scanning techniques and technologies will have to empower their engines with powerful client-side DOM related vulnerability detection mechanisms. It may be a challenge to perform complete automated scanning for Web 2.0 applications, one that can be

surmounted by using automated scanning in combination with human intelligence. Some of the old attack vectors such as Cross-Site Request Forgery (XSRF) [ref 10], are also being looked at afresh in this era of Web 2.0. XSS, XSRF and other client-side attacks feature in several vulnerabilities and advisories for new generation Web applications.

Web 2.0 application assessment needs special attention to be devoted to client-side attack vectors with the purpose of mitigating risks at client ends. This article has sought to throw light on some attack vectors and scanning techniques used to identify vulnerable applications. Scratch the surface and many applications that are vulnerable to this range of attack vectors and that can be exploited by application layer attackers, viruses and worms, will be revealed.

References

[ref 1] Brief on mashup (http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid))
[ref 2] JavaScript Object Notation (JSON) is a lightweight data-interchange format (http://www.json.org/)
[ref 3] Hacking Web 2.0 Applications with Firefox (http://www.securityfocus.com/infocus/1879)
[ref 4] XSS threat classification (http://www.webappsec.org/projects/threat/classes/cross-site_scripting.shtml)
[ref 5] DOM Based Cross Site Scripting or XSS of the Third Kind - By Amit Klein(http://www.webappsec.org/projects/articles/071105.shtml)
[ref 6] Web developer plugin (http://chrispederick.com/work/webdeveloper/)
[ref 7] Dojo toolkit (http://www.dojotoolkit.com/)
[ref 8] JSON callback (http://developer.yahoo.com/common/json.html)
[ref 9] The Web Hacking Incidents Database (http://www.webappsec.org/projects/whid/)
[ref 10] Cross-Site Reference Forgery - By Jesse Burns (http://www.isecpartners.com/documents/XSRF_Paper.pdf

About the author

Shreeraj Shah, B.E., MSCS, MBA, is the founder of Net Square and leads Net Square's consulting, training and R&D activities. Prior to founding Net-Square, he has 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 online.

Reprints or translations

Reprint or translation requests require prior approval from SecurityFocus.

© 2006 SecurityFocus

Comments?

Public comments for Infocus technical articles, as shown 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