Top 10 AJAX security holes & driving factors

Shreeraj Shah
Founder, Net Square
shreeraj@net-square.com

Introduction

One of the central ingredients of Web 2.0 applications is Ajax encompassed by JavaScripts. This phase of evolution has transformed the Web into a superplatform. Not surprisingly, this transformation has also given rise to a new breed of worms and viruses such as Yamanner, Samy and Spaceflash. Portals like Google, NetFlix, Yahoo and MySpace have witnessed new vulnerabilities in the last few months. These vulnerabilities can be leveraged by attackers to perform Phishing, Cross-site Scripting (XSS) and Cross-Site Request Forgery (XSRF) exploitation.

There is no inherent security weakness in Ajax but adaptation of this technology vector has changed the Web application development approach and methodology significantly. Data and object serialization was very difficult in the old days when DCOM and CORBA formed the core middleware tier. Ajax can consume XML, HTML, JS Array, JSON, JS Objects and other customized objects using simple GET, POST or SOAP calls; all this without invoking any middleware tier. This integration has brought about a relatively seamless data exchange between an application server and a browser. Information coming from the server is injected into the current DOM context dynamically and the state of the browser's DOM gets recharged. Before we take a look at security holes let's examine the key factors that seem to be driving Web 2.0 vulnerabilities.

- ***Multiple scattered end points and hidden calls*** – One of the major differences between Web 2.0 applications and Web 1.0 is the information access mechanism. A Web 2.0 application has several endpoints for Ajax as compared to its predecessor Web 1.0. Potential Ajax calls are scattered all over the browser page and can be invoked by respective events. Not only does this scattering of Ajax calls make it difficult for developers to handle, but also tends to induce sloppy coding practices given the fact that these calls are hidden and not easily obvious.

- ***Validation confusion*** – One of the important factors in an application is input and outgoing content validation. Web 2.0 applications use bridges, mashups, feeds, etc. In many cases it is assumed that the "other party" (read server-side or client-side code) has implemented validation and this confusion leads to neither party implementing proper validation control.

- ***Untrusted information sources*** – Web 2.0 applications fetch information from various untrusted sources such as *feeds*, *blogs*, *search* results. This content is never validated prior to being served to the end browser, leading to cross-site exploitation. It is also possible to load JavaScript in the browser that forces the browser to make cross-domain calls and opens up security holes. This can be lethal and leveraged by virus and worms.

- ***Data serialization*** **–** Browsers can invoke an Ajax call and perform data serialization. It can fetch JS array, Objects, Feeds, XML files, HTML blocks and JSON. If any of these serialization blocks can be intercepted and manipulated, the browser can be forced to execute malicious scripts. Data serialization with untrusted information can be a lethal combination for end-user security.

- ***Dynamic script construction & execution*** **–** Ajax opens up a backend channel and fetches information from the server and passes it to the DOM. In order to achieve this one of the requirements is the dynamic execution of JavaScripts to update the state of the DOM or the browser's page memory. This is achieved by calling customized functions or the eval() function. The consequence of not validating content or of making an insecure call can range from a session compromise to the execution of malicious content.

Web 2.0 applications can become vulnerable with one or more lapses mentioned above. If developers have not taken enough precautions in putting in place security controls, then security issues can be opened up on both the server as well as browser ends. Here is a list and brief overview of ten possible security holes.

(1) Malformed JS Object serialization

JavaScript supports Object-Oriented Programming (OOP) techniques. It has many different built-in objects and allows the creation of user objects as well. A new object can be created using *new object()* or simple inline code as shown below:

```
message = {
      from : "john@example.com",
      to : "jerry@victim.com",
      subject : "I am fine",
      body : "Long message here",
      showsubject : function(){document.write(this.subject)}
};
```

Here is a simple message object that has different fields required for email. This object can be serialized using Ajax and consumed by JavaScript code. The programmer can either assign it to the variable and process it or make eval(). If an attacker sends a malicious "subject" line embedded with script then it makes the reader a victim of cross-site scripting attacks. A JS object can have both data and methods. Improper usage of JS object serialization can open up a security hole that can be exploited by crafty packet injection code.

(2) JSON pair injection

JavaScript Object Notation (JSON) is a simple and effective lightweight data exchange format and one that can contain object, array, hash table, vector and list data structures. JSON is supported by JavaScript, Python, C, C++, C# and Perl languages. Serialization of JSON is a very effective exchange mechanism in Web 2.0 applications. Developers choose JSON over Ajax very frequently and fetch and pass required information to the DOM. Here is a simple JSON object “bookmarks” object with different name-value pair.

```
{"bookmarks":[{"Link":"www.example.com","Desc":"Interesting link"}]}
```

It is possible to inject a malicious script in either `Link` or `Desc`. If it gets injected into the DOM and executes, it falls into the XSS category. This is another way of serializing malicious content to the end-user.

(3) JS Array poisoning

JS array is another very popular object for serialization. It is easy to port across platforms and is effective in a cross-language framework. Poisoning a JS array spoils the DOM context. A JS array can be exploited with simple cross-site scripting in the browser. Here is a sample JS array:

```
new Array(“Laptop”, “Thinkpad”, “T60”, “Used”, “900$”, “It is great and I
have used it for 2 years”)
```

This array is passed by an *auction site* for a *used laptop*. If this array object is not properly sanitized on the server-side, a user can inject a script in the last field. This injection can compromise the browser and can be exploited by an attack agent.

(4) Manipulated XML stream

An Ajax call consumes XML from various locations. These XML blocks originate from Web services running on SOAP, REST or XML-RPC. These Web services are consumed over proxy bridges from third-parties. If this third-party XML stream is manipulated by an attacker then the attacker can inject malformed content.

The browser consumes this stream from its own little XML parser. This XML parser can be vulnerable to different XML bombs. It is also possible to inject a script in this stream which can again, lead to cross-site scripting (XSS). XML consumption in the browser without proper validation can compromise the end-client.

(5) Script injection in DOM

The first four holes were the result of issues with serialization. Once this serialized stream of object is received in the browser, developers make certain calls to access the DOM. The objective is to “repaint” or “recharge” the DOM with new content. This can be done by calling *eval(),* a customized function or *document.write()*. If these calls are made on untrusted information streams, the browser would be vulnerable to a DOM manipulation vulnerability. There are several *document.*()* calls that can be utilized by attack agents to inject XSS into the DOM context.

For example, consider this line of JavaScript code, `Document.write(product-review)`

Here, "`Product-review`" is a variable originating from a third-party blog. What if it contains JavaScript? The answer is obvious. It will get executed in the browser.

(6) Cross-domain access and Callback

Ajax cannot access cross-domains from the browser. One of the browser security features that exists in all flavors of browsers is the blocking of cross-domain access. There are several Web services that provide a callback mechanism for object serialization. Developers can use this callback mechanism to integrate Web services in the browser itself. The callback function name can be passed back so that as soon as the callback object stream is retrieved by the browser it gets executed by the specific function name originally passed from the browser.

This callback puts an extra burden on developers to have in-browser validation. If the incoming object stream is not validated by the browser then developers are putting the end client's fate at the mercy of cross-domain targets. Intentionally or unintentionally, this cross domain service can inject malicious content into the browser. This cross domain call runs in the current DOM context and so makes the current session vulnerable as well. This entire cross-domain mechanism needs to be looked at very closely before implementation into an application.

(7) RSS & Atom injection

Syndicated feeds, RSS and Atom, are one of the most popular ways of passing site-updated information over the Internet. Several news, blogs, portals, etc. share more than one feed over the Internet. A *feed* is a standard XML document and can be consumed by any application. Web 2.0 applications integrate syndicated feeds using widgets or in-browser components. These components make Ajax calls to access feeds.

These feeds can be selected by end-users easily. Once selected, these feeds are parsed and injected into the DOM. But if the feed is not properly validated prior to injecting it into the DOM, several security issues can crop up. It is possible to inject a malicious link or JavaScript code into the browser. Once this malicious code injected into the DOM, the game is over. The end result is XSS and session hijacking.

(8) One-click bomb

Web 2.0 applications may not be compromised at the first instance itself, but it is possible to make an event-based injection. A malicious link with "onclick" can be injected with JavaScript. In this case, the browser is sitting on an exploit bomb waiting for the right event from the end-user to trigger the bomb. The exploit succeeds if that particular event is fired by clicking the link or button. This can lead to session hijacking through malicious code.

Once again this security hole is opened up as a result of information processing from untrusted sources without the right kind of validation. To exploit this security hole an event is required to be fired from an end-client. This event may be an innocuous event such as *clicking a button or a link* but the consequences can be disastrous. A malicious event that is fired may send current session information to the target or execute fancy inline exploit scripts in current browser context.

(9) Flash-based cross domain access

It is possible to make GET and POST requests from JavaScripts within a browser by using a Flash plugin's Ajax interface. This also enables cross-domain calls to be made from any particular domain. To avoid security concerns, the Flash plugin has implemented policy-based access to other domains. This policy can be configured by placing the file *crossdomain.xml* at the root of the domain. If this file is left poorly configured – as is quite often the case – it opens up the possibility of cross-domain access. Here is a sample of a poorly configured XML file:

```
<cross-domain-policy>
    <allow-access-from domain="*"/>
</cross-domain-policy>
```

Now, it is possible to make cross-domain calls from within the browser itself. There are a few other security issues concerning this framework as well. Flash-based Rich Internet Applications (RIA) can be vulnerable to a cross-domain access bug over Ajax if deployment is incorrect.

(10) XSRF

Cross-Site Request Forgery is an old attack vector in which a browser can be forced to make HTTP GET or POST requests to cross-domains; requests that may trigger an event in the application logic running on the cross-domain. These can be requests for a change of password or email address. When the browser makes this call it replays the cookie and adopts an identity. This is the key aspect of the request. If an application makes a judgment on the basis of cookies alone, this attack will succeed.

In Web 2.0 applications Ajax talks with backend Web services over XML-RPC, SOAP or REST. It is possible to invoke them over GET and POST. In other words, it is also possible to make cross-site calls to these Web services. Doing so would end up compromising a victim's profile interfaced with Web services. XSRF is an interesting attack vector and is getting a new dimension in this newly defined endpoints scenario. These endpoints may be for Ajax or Web services but can be invoked by cross-domain requests.

Exploitation of security holes and Countermeasures

Web 2.0 applications have several endpoints; each an entry point for threat modeling. To provide proper security it is imperative to guard each of these entry points. Third-party information must be processed thoroughly prior to sending it to the end-client.

To deal with Ajax serialization issues validation must be placed on incoming streams before they hit the DOM. XML parsing and cross-domain security issues need extra attention and better security controls. Follow the simple thumb rule of not implementing cross-domain information processing into the browser without proper validation. Interestingly, up until now, the use of client-side scripts for input validation was thoroughly discouraged by security professionals because they can be circumvented easily.

Web 2.0 opens up several new holes around browser security. Exploitation of these security holes is difficult but not impossible. Combinations of security issues and driving factors can open up exploitable holes that impact the sizeable Web community, such as those that can be leveraged by attackers, worms and viruses. Identity compromise may be the final outcome.

Conclusion

This article has briefly touched upon a few likely security holes around Ajax. There are a few more lurking around, such as the ones leveraging cross-domain proxies to establish a one-way communication channel or memory variable access in the browser.

With Web 2.0, a lot of the logic is shifting to the client-side. This may expose the entire application to some serious threats. The urge for data integration from multiple parties and untrusted sources can increase the overall risk factor as well: XSS, XSRF, cross-domain issues and serialization on the client-side and insecure Web services, XML-RPC and REST access on the server-side. Conversely, Ajax can be used to build graceful applications with seamless data integration. However, one insecure call or information stream can backfire and end up opening up an exploitable security hole.

These new technology vectors are promising and exciting to many, but even more interesting to attack, virus and worm writers. To stay secure, this is all the more reason for developers to paying attention to implementation detail.