The security risks of AJAX/web 2.0 applications

Paul Ritchie, security consultant, SecureTest Ltd

Paul Ritchie

The term 'web 2.0' was coined by O'Reilly Media following a number of conferences that it hosted in 2004. The popular media latched onto the concept and turned it into a popular phrase that has become synonymous with a new breed of website. Web 2.0 sites typically bring user collaboration to the foreground and offer interactivity closer to that of a desktop application.

The web 2.0 concept is not sufficiently defined to allow a critical discussion of it, but we can discuss AJAX (asynchronous Javascript and XML). This concept, which underpins many web 2.0 sites, repackages and applies existing technologies to achieve a new structure for internet applications. Unfortunately, increased flexibility creates conditions for new security problems.

To understand how AJAX alters the security landscape for web application testing it is necessary to show the fundamental differences between it and traditional internet application models.

Traditional internet model

A web browser requests a webpage, normally indicating that the request is being processed by animating a logo and altering the status bar. Internet Explorer, for example, animates the chequered flag. Figure 1 summarises this interaction.

When the user clicks on a link, an HTTP Get request is sent to the server. The web server deals with the request, and sends the web page to the client. If the client is to send information back to the server, another request is made following the same process.

Under this synchronous 'click-and-wait' communication method, information is exchanged by requesting and receiving whole web pages. While waiting for the server, the user loses the focus of the application and cannot interact with it. This loss of focus has long been a source of dissatisfaction with traditional web applications, and if the wait for a round trip from the server is sufficiently long, users may leave the site.

The AJAX-enabled internet model

In this model a client requests a webpage. Once this full page is loaded, communication between the client and the server can be conducted in an asynchronous manner. This minimises the client's waiting time, because only partial user interface update requests are made.

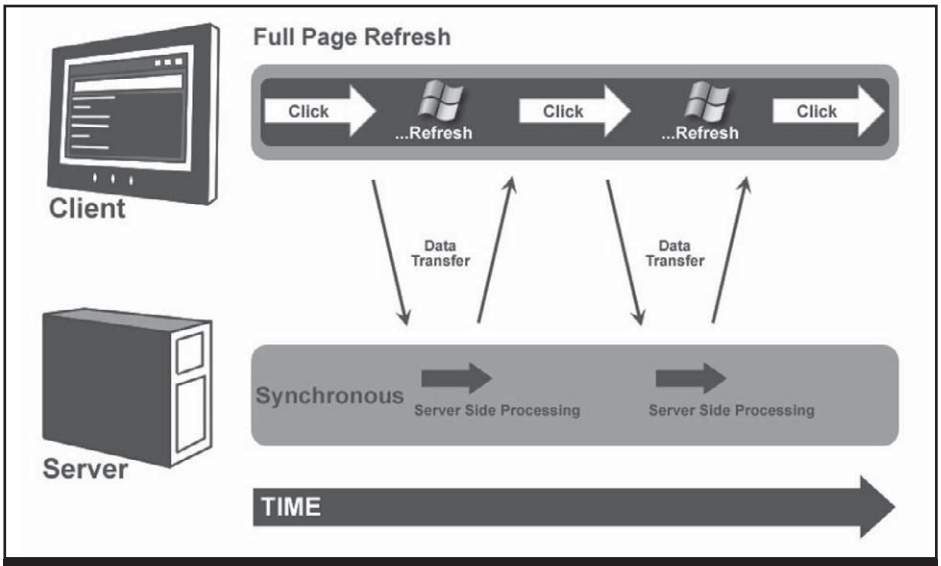

Figure 1: Traditional synchronous model for the internet

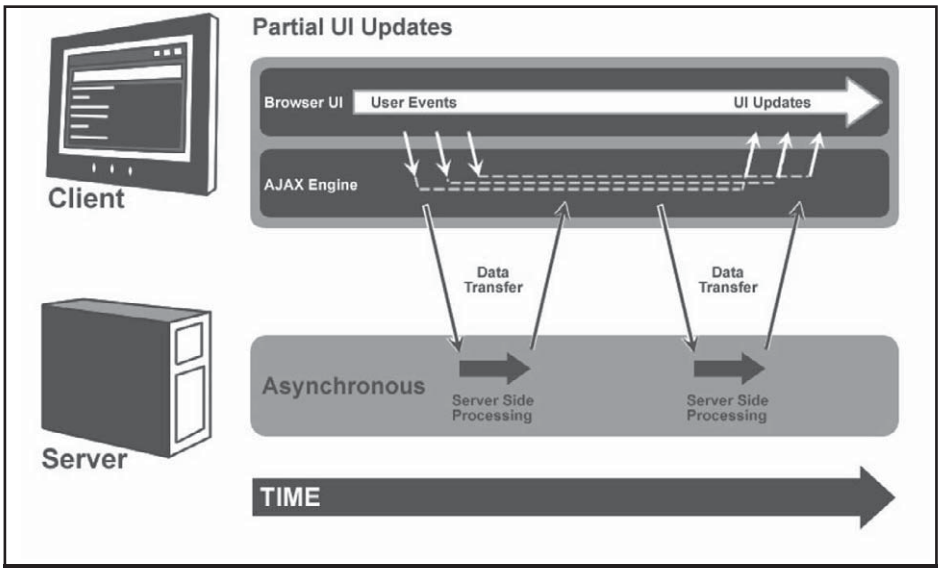

Figure 2: AJAX-enabled asynchronous internet model

Only aspects of the client's user interface are updated in an AJAX scenario. Those that are not modified by the user remain static, reducing the communication overhead. This leaves the focus of the application with the user, creating a feeling of seamless interactivity.

Figure 2 shows this style of communication.

The AJAX engine is the client side code that handles calls between the client and server. Typically this would be a library of Javascript functions included on the page.

While this is a generally accepted terminology, it can be confusing because it is not standard across AJAX applications. A more appropriate term might be 'application logic'.

Underlying scripts and protocols

AJAX is nothing new per se as it is based on technologies that have existed for many years. AJAX applications use the following components to achieve this asynchronous interaction: a client-side scripting language such as Javascript, an XmlHttpRequest object provided by the web browser, and a response data format.

The XmlHttpRequest object provides an API for making server requests via HTTP, with the chosen scripting language making the appropriate calls. The client-side scripting language needs to be able to access the XmlHttpRequest object provided by the web browser and interpret its responses appropriately. Many client-side scripting languages are capable of this, but typically Javascript is used since most browsers support it. The data format returned from the server is entirely up to the developer, but it is usual for XML, plain text or HTML to be used.

The fact that all web developers are familiar with Javascript and HTML can explain the rapid adoption of the AJAX approach because the learning curve was sufficiently low.

With these basic models defined it is possible to discuss some of the categories of security problems most applicable to an AJAX - enabled web application.

An article from www.securityfocus.com, *Ajax Security Basics*[1], points out three important ways in which an AJAX-enabled application can introduce more, or different, security vulnerabilities than a traditional synchronous request/response application. These are client-side security controls, increased attack surfaces, and new possibilities for Cross-Site Scripting (XSS).

Penetration testers have been advising for years that all input validation on the client side should be replicated on the server side. This is because any attacker with basic skills can use proxy software (or call script functions directly) to bypass the intended logic.

While this is not a new security concept, AJAX potentially increases the reliance on client technologies for security measures, since more application logic is being delegated to web browsers. This allows intruders to easily read the source code and look for areas of weakness.

Client-side security control vulnerability example

Here is a simple example of an application using the AJAX structure to permit a login request. Assume that the Javascript in listing 1 appears on a simple login page. The function calls in bold are calls to locally defined functions.

The user types in their username and password as normal but when the submit button is clicked a Javascript command (doLogin) is called instead of a request being made directly to the server. The doLogin function ensures that the parameters match a valid format before the login request is made using the XMLHttpRequest object.

The server's response is passed to the handleResponse Javascript function when it arrives. It is in this function that the problem exists and can be summarised with listing 3.

When the login has failed the message is displayed as a Javascript alert but when the login has succeeded the loginUser function is called.

> "Client-side security control is a well-known problem when developers rely on client technology for anything more than reducing spurious requests to the server."

It is logically the correct behaviour but because this has been located as part of the client - side AJAX system an attacker can simply call the loginUser request themselves which bypasses the server check. This can be achieved by typing directly into the URL bar of a web browser, as in figure 3.

This issue is a consequence of moving critical security logic to the client's computer, showing how client-side security measures can be bypassed. Developers should use the server-side application to implement all security procedures, keeping the application logic further away from intruders.

The second general threat category mentioned by the securityfocus.com article[1] is increased attack surface. AJAX encourages developers to create their applications in smaller chunks than conventional web applications. Traditionally, one page would serve multiple smaller tasks to compensate for the network overheads.

A more granular application design increases the number of distinct AJAX endpoints on the system, which is logically analogous to a host offering more open TCP ports to network layer attackers. Each additional AJAX endpoint both increases the complexity of development and the likelihood that a necessary security control is neglected.

For example, consider a common practice where a PHP file is set as required for inclusion by a set of web pages[2]. This

Listing 1: Vulnerable client-side login script

```
<script type="text/javascript" language="javascript">
/* Get an XMLHttpRequest Object for most web browsers
   cleanly and return it */
function getXMLhttpRequest() {
        // Code has been removed since it is unimportant
        return xmlhttp ;
}
/* Ensures that user and pass are correctly formatted */
function validateLogin(user, pass) {
        // Code has been simplified
        if(invalid) { return false ; }
        return true ;
}
/* This is called when the form submits */
function doLogin() {
        var user = document.getElementById("user").value ;
        var pass = document.getElementById("pass").value ;
        // validate params.
        valid = validateLogin(user, pass) ;
        if (valid==true) {
                // Parameters formatted correctly
                var xmlhttp = getXMLhttpRequest() ;
                xmlhttp.onreadystatechange
= function() { handleResponse(xmlhttp); };
                xmlhttp.open('GET',
 "login.php?user="+user+"&pass="+pass,
true);
                xmlhttp.send(null);
        } else {
                // Parameters invalidly formatted
                alert("Username or password were not correctly formatted");
        }
}
/* Handles the login request's response */
function handleResponse(xmlhttp) {
        if (xmlhttp.readyState == 4) {
                if (xmlhttp.status == 200) {
                        var response = xmlhttp.responseText ;
                        if (response != "Login Successful") {
                                // Login Failed
                                alert(response);
                        } else {
                                // Login Succeeded
                                loginUser() ;
                        }
                } else {
                        alert('There was a problem with the request.');
                }
        }
}
/* Called when supplied credentials were valid */
function loginUser() {
        var user = document.getElementById("user").value ;
        var xmlhttp = getXMLhttpRequest() ;
        xmlhttp.onreadystatechange
= function() { updateContent(xmlhttp); };
        xmlhttp.open('GET', "createSession.php?user="+user, true);
        xmlhttp.send(null);
}
/* Updates part of the page to show the authenticated content */
function updateContent(xmlhttp) {
        if (xmlhttp.readyState == 4) {
                if (xmlhttp.status == 200) {
                        var response = xmlhttp.responseText ;


                        document.getElementById("content").innerHTML
= response ;
                } else {
                        alert('There was a problem with the request.');
                }
}
   }
</script>
```

could be used to ensure users are authenticated to the system. A developer could easily forget to set this option when adding a new page, and the chances can only increase as the number of pages increases.

The more components a system has, the more complex it will become to maintain, manage, and update. In turn this increases the chances of critical flaws going unnoticed by internal auditing.

SecureTest has seen a case where this added complexity has led to a potential security vulnerability. While testing an AJAX-enabled application with multiple levels of user account it was found that the site employed one Javascript include file for the entire client-side logic.

This meant that an anonymous user with a trial account could see the logic behind an administrator-level service call. The locations of all the administrator service scripts were disclosed, providing a definitive map of the application to a potential attacker.

It can be argued that this is a direct result of system complexity because the developers have attempted to simplify things by putting all AJAX-style service calls in one file for editing reasons. The only effective guard against this category is to ensure a good policy for peer code review and to implement effective procedures for the application of all the necessary security checks in any new code.

Cross-site scripting

The final point from the securityfocus.com article concerns the new

Listing 2: HMTL form calling listing 1

```
<html>
…
<div id="content">
<form action="javascript:doLogin()">
        <input type="text" name="user">
        <input type="password" name="pass">
        <input type="submit"/>
</form>
</div>
…
</html>
```

Listing 3: Offending subsection of handle Response function

```
if (response != "Login Successful") {
// Login Failed
        alert(response);
} else {
        // Login Succeeded
        loginUser() ;
}
```

possibilities for cross-site scripting (XSS). This has recently been the most fertile ground for the exploitation of AJAX-enabled web applications in the real world leading to the high profile Samy[3] worm for myspace.com. XSS is the injection of HTML and client-side scripting (i.e. Javascript) into a page that is returned to the user's browser. Typically this is possible where an HTTP Get parameter is accepted without proper input validation checks and then echoed back to the user.

In this case a clickable URL can be created which contains maliciously embedded script commands. The attacker emails this link to a victim and when the page loads the malicious scripting is executed and an attacker can potentially hijack a session (access the victim's account by stealing cookies). Other possibilities include creating a fake login (to steal credentials) or logging keystrokes (to steal credentials).

Figure 3: Execute 'loginuser' directly from address bar

XSS is a common problem on the Internet which is often a core component of phishing attacks. In an AJAX application the dangers of XSS actually increase for a number of reasons. Firstly XSS lasts as long as the affected page is loaded. Since, in theory, only one page is loaded in an AJAX application there is potential to create a permanent XSS issue throughout an entire user session.

> "XSS has recently been the most fertile ground for the exploitation of AJAX-enabled web applications in the real world, leading to the high profile Samy worm for myspace.com."

In a traditional web application the XSS would typically be short lived as clicking on any link could effectively end the attack. This has led to a class of limited exploitation with a one-shot payload. The exploit had to execute immediately on clicking. An XSS attack in an AJAX style application enables the attacker to potentially continue the exploit in many more ways, including logging of keystrokes across the whole session.

An attacker may also issue requests to the server which are completely hidden from the victim. Internet Explorer's chequered flag and status bar do not alter when a request is sent through the XMLHttpRequest object.

This means that the victim has no visual cue that something malicious is happening since they still have the focus of the web application to continue their intended interactions. This is also the case with Mozilla Firefox, Opera, and Safari, which all show no visual cues when an XMLHttpRequest object is used.

Exploiting XSS using traditional vectors like a hidden iframe (an HTML element which allows the embedding of another HTML document inside the main document) or a window.location redirect (typically with cookie values appended to the request) could result in visual cues that would make a victim suspicious.

The Samy worm

The Samy worm was based on defeating the input validation controls with a clever iterative process to figure out how to get certain commands echoed to the user's web browser. Effectively this is a classic XSS vulnerability but it differed by achieving a seamless self-propagating exploit of the myspace.com website.

It worked as follows:

1) A victim visits an infected myspace.com profile (originally this would have been samy himself)
2) Javascript is used to get the User ID of the victim from the HTML source using DOM (Document Object Model)
3) An AJAX request is used to get the victim's friend list
4) Adds a friend called "samy" to the victim's friend list (with the message "but most of all, samy is my hero.")

The friend profile for samy, which is now on the victim's profile page, includes the code to infect people who view their profile. This is a very good example of the potential for an XSS vulnerability in an AJAX environment to more seamlessly infect a whole site rapidly. If the worm had been chosen to deliver another payload (i.e. add the friend and then log a user out) there would have been an effective denial of service condition.

Cutting edge AJAX vulnerabilities

The 23rd CCC (Chaos Communication Congress) in Berlin,

December 2006, saw the release of a paper entitled *Subverting Ajax* by Stefano Di Paola and Giorgio Fedon[4]. The paper discusses an interesting concept that Di Paola calls prototype hijacking, which exploits the design of the Javascript language.

While Javascript is object-oriented, it is based on prototypes. All objects are simply clones of prototyped original objects and it is possible to override any member variable or function.

For example it is possible for code inserted via an XSS vulnerability to create a wrapper of the original XMLHttpRequest object which would allow the attacker to monitor the legitimate traffic remotely, leaving the victim unaware. News of this issue caused a stir among the security community in early 2007 with many people claiming it is a flaw with Javascript. Prototype-based programming is a design choice which is a very useful feature of Javascript and it is unlikely to be removed.

The only effective solution is to prevent the XSS flaws from happening by improving input validation techniques for web applications and ensuring that any user input is HTML encoded before being echoed back to a user's screen.

Conclusion

We are entering a new development phase for internet applications which is just being understood by security researchers. It is expected that there are plenty of potential issues with AJAX yet to be discovered but this is no reason to be scared. User satisfaction is a worthy goal and an asynchronous communication model can be the solution for web applications.

Potentially the biggest threat comes from XSS, which can be used in new and more dangerous ways. However, XSS attacks can be avoided using more stringent input validation and encoding. The solution to all the problems discussed in this article is due care and regular auditing.

References

1. Hayre and Kelath, "Ajax Security Basics", Security Focus, 22 June 2006. Symantec. <www.securityfocus.com/infocus/1868>.
2. Entry for 'require' command, PHP online manual, <http://ca3.php.net/require>
3. Technical explanation of the Samy MySpace worm by the author, 2005 <http://namb.la/popular/tech.html>
4. S. Di Paola and G. Fedon, Subverting Ajax, 23rd Chaos Communication Congress, Dec 2006 <http://events.ccc.de/congress/2006/Fahrplan/attachments/1158-Subverting_Ajax.pdf>

Resources

Andrew Sutherland, Periodic table of the elements (AJAX demonstration). <http://code.jalenack.com/periodic/>

Coach Wei, Ajax - Asynchronous Java + XML?, CoachWei blog. <www.coachwei.com/blog/_archives/2005/8/14/1135700.html>

About the author

Paul Ritchie is a penetration tester for SecureTest Ltd. He is based in the UK. He has a BSc (Hons) in Computer Science and an MSc in ecommerce - both from the University of Aberdeen. Paul's main professional focus is web application security, where he has been the lead consultant in many engagements for clients ranging from UK local government to multinationals.

Maximizing the ROI of a security audit

Ross Westcott, Chief IT auditor, Portland General Electric Company

Ross Westcott

Audits generally cause IT security managers to pause and wonder if their house is in order enough to survive the effort. Having someone look over your work may be nerve-wracking, but it is unarguably constructive, and information security managers are learning to anticipate rather than fear this event. It is an opportunity to highlight what is being done correctly, and to identify areas that need improvement.

"The mature information security program is one that takes advantage of all resources available," says security guru Thomas Peltier of security consultancy Peltier Associates. "The security team and the audit staff provide a formidable force that can ensure the integration and acceptance of an enterprise-wide information security program." A long-term proposition of a fully functional security program is a long-term proposition. No one can gain assurance through a one-time checklist or a single penetration attempt. On the contrary, ensuring adequate security is a multi-year, multi-faceted process. If the bad guys want into an organisation bad enough, they will not stop at one roadblock and will continue to develop their tools and techniques. As attacks evolve and change, so must asset protection schemes and audits.

Audits are sometimes benign, finding very little of substance. This can mean one of three things. Either all is well in the security world, the audit did not look