

# JavaScript Hijacking

Brian Chess, Yekaterina Tsipenyuk O'Neil, Jacob West

{brian, katrina, jacob}@fortifysoftware.com

March 12, 2007

## Summary

An increasing number of rich Web applications, often called Ajax applications, make use of JavaScript as a data transport mechanism. This paper describes a vulnerability we term JavaScript Hijacking, which allows an unauthorized party to read confidential data contained in JavaScript messages. The attack works by using a `<script>` tag to circumvent the Same Origin Policy enforced by Web browsers. Traditional Web applications are not vulnerable because they do not use JavaScript as a data transport mechanism.

We analyzed 12 popular Ajax frameworks, including 4 server-integrated toolkits – Direct Web Remoting (DWR), Microsoft ASP.NET Ajax (a.k.a. Atlas), xajax and Google Web Toolkit (GWT) -- and 8 purely client-side libraries -- Prototype, Script.aculo.us, Dojo, Moo.fx, jQuery, Yahoo! UI, Rico, and MochiKit. We determined that among them only DWR 2.0 implements mechanisms for preventing JavaScript Hijacking. The rest of the frameworks do not explicitly provide any protection and do not mention any security concerns in their documentation.

Many programmers are not using any of these frameworks, but based on our findings with the frameworks, we believe that many custom-built applications are also vulnerable. An application may be vulnerable if it:

- Uses JavaScript as a data transfer format
- Handles confidential data

We advocate a two-pronged mitigation approach that allows applications to decline malicious requests and prevent attackers from directly executing JavaScript the applications generate.

## 1. Introduction

Although the term “Web 2.0” does not have a rigorous definition, it is commonly used in at least two ways. First, it refers to Web applications that encourage social interaction or collective contribution for a common good. Second, it refers to Web programming techniques that lead to a rich and user-friendly interface. These techniques sometimes go by the name Asynchronous JavaScript and XML (Ajax), though many implementations use no XML at all. In some cases, the social and technical aspects of Web 2.0 come together in the form of *mashups*: Web applications that are built by assembling pieces from multiple independent Web applications.

This paper describes a vulnerability we term *JavaScript Hijacking*. It is an attack against the data transport mechanism used by many rich Web applications. JavaScript Hijacking allows an unauthorized attacker to read confidential data from a vulnerable application using a technique similar to the one commonly used to create mashups. The vulnerability is already being discussed

in some circles<sup>1</sup>, but the majority of Web programmers are not aware that the problem exists, and even fewer security teams understand how widespread it is.

Traditional Web applications are not vulnerable to JavaScript Hijacking because they do not use JavaScript as a data transport mechanism. To our knowledge, this is the first class of vulnerability that is specific to rich Web applications. In essence, JavaScript Hijacking is possible because the security model implemented by all popular web browsers does not anticipate the use of JavaScript for communicating confidential information.

JavaScript Hijacking builds upon another type of widespread vulnerability: cross-site request forgery. A cross-site request forgery attack causes a victim to unwittingly submit one or more HTTP requests to a vulnerable website. A typical cross-site request forgery attack compromises data integrity—it gives an attacker the ability to modify information stored by a vulnerable website. JavaScript Hijacking is more dangerous because it also compromises confidentiality—an attacker can read a victim's information.

Vulnerable websites have already been found in the wild. One of the first people to demonstrate JavaScript Hijacking was Jeremiah Grossman, who identified a vulnerability in Google GMail.<sup>2</sup> (Google has fixed the problem.) Google was serving the current GMail users' contacts in unprotected JavaScript, so an attacker could steal the contact list using JavaScript Hijacking.

During the course of our work, we examined 12 popular Ajax frameworks, including 4 server-integrated toolkits and 8 purely client-side libraries. We found that only 1 of the 12 took measures to prevent JavaScript Hijacking. Preventing JavaScript Hijacking requires a secure server-side implementation, but it is incumbent upon the client-side libraries to promote good security practices. Currently, some of the client-side libraries go so far as to require the server side to contain a JavaScript Hijacking vulnerability.

Section 2 describes JavaScript Hijacking and explains why rich Web applications are vulnerable while older Web applications are not. Section 3 looks at methods for defending against JavaScript Hijacking. Section 4 discusses popular Ajax frameworks and explains which ones are vulnerable.

## **2. JavaScript Hijacking**

Web browsers enforce the Same Origin Policy in order to protect users from malicious websites. The Same Origin Policy requires that, in order for JavaScript to access the contents of a Web page, both the JavaScript and the Web page must originate from the same domain. Without the Same Origin Policy, a malicious website could serve up JavaScript that loads sensitive information from other websites using a client's credentials, culls through it, and communicates it back to the attacker.

JavaScript Hijacking allows an attacker to bypass the Same Origin Policy in the case that a Web application uses JavaScript to communicate confidential information. The loophole in the Same Origin Policy is that it allows JavaScript from any website to be included and executed in the context of any other website. Even though a malicious site cannot directly examine any data loaded from a vulnerable site on the client, it can still take advantage of this loophole by setting up an environment that allows it to witness the execution of the JavaScript and any relevant side effects it may have. Since many Web 2.0 applications use JavaScript as a data transport mechanism, they are often vulnerable while traditional Web applications are not.

---

<sup>1</sup> [http://getahead.org/blog/joe/2007/03/05/json\\_is\\_not\\_as\\_safe\\_as\\_people\\_think\\_it\\_is.html](http://getahead.org/blog/joe/2007/03/05/json_is_not_as_safe_as_people_think_it_is.html)

<sup>2</sup> <http://jeremiahgrossman.blogspot.com/2006/01/advanced-Web-attack-techniques-using.html>

The most popular format for communicating information in JavaScript is *JavaScript Object Notation* (JSON). The JSON RFC defines JSON syntax to be a subset of JavaScript object literal syntax<sup>3</sup>. JSON is based on two types of data structures: arrays and objects. Any data transport format where messages can be interpreted as one or more valid JavaScript statements is vulnerable to JavaScript Hijacking. JSON makes JavaScript Hijacking easier by the fact that a JSON array stands on its own as a valid JavaScript statement. Since arrays are a natural form for communicating lists, they are commonly used wherever an application needs to communicate multiple values. Put another way, a JSON array is directly vulnerable to JavaScript Hijacking. A JSON object is only vulnerable if it is wrapped in some other JavaScript construct that stands on its own as a valid JavaScript statement.

The following example begins by showing a legitimate JSON interaction between the client and server components of a Web application that is used to manage sales leads. It goes on to show how an attacker can mimic the client and gain access to the confidential data the server returns. Note that this example, along with the rest of the examples in the paper, is written for Mozilla-based browsers. Other mainstream browsers do not allow native constructors to be overridden when an object is created without the use of the `new` operator.

The client requests data from a server and evaluates<sup>4</sup> the result as JSON with the following code:

```
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json", true);
req.onreadystatechange = function () {
    if (req.readyState == 4) {
        var txt = req.responseText;
        object = eval("(" + txt + ")");
        req = null;
    }
};
req.send(null);
```

When the code runs, it generates an HTTP request that looks like this:

```
GET /object.json HTTP/1.1
...
Host: www.example.com
Cookie: JSESSIONID=F2rN6HopNzsfXFjHX1c5Ozxi0J5SQZTr4a5YJaSbAiTnRR
```

(In this HTTP response and the one that follows we have elided HTTP headers that are not directly relevant to this explanation.)

The server responds with an array in JSON format:

```
HTTP/1.1 200 OK
Cache-control: private
```

---

<sup>3</sup> <http://www.ietf.org/rfc/rfc4627.txt>

<sup>4</sup> The code we have examined uses `eval()` to evaluate JSON. A more secure alternative is to use `parseJSON()`, which can help prevent some cross-site scripting attacks by accepting only JSON syntax.

Content-Type: text/javascript; charset=utf-8

...

```
[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
  "purchases":60000.00, "email":"brian@fortifysoftware.com" },
 {"fname":"Katrina", "lname":"O'Neil", "phone":"6502135600",
  "purchases":120000.00, "email":"katrina@fortifysoftware.com" },
 {"fname":"Jacob", "lname":"West", "phone":"6502135600",
  "purchases":45000.00, "email":"jacob@fortifysoftware.com" }]
```

In this case, the JSON contains confidential information associated with the current user (a list of sales leads). Other users cannot access this information without knowing the user's session identifier. (In most modern Web applications, the session identifier is stored as a cookie.) However, if a victim visits a malicious website, the malicious site can retrieve the information using JavaScript Hijacking.

If a victim can be tricked into visiting a Web page that contains the following malicious code, the victim's lead information will be sent to the attacker's Web site.

```
<script>
// override the constructor used to create all objects so
// that whenever the "email" field is set, the method
// captureObject() will run. Since "email" is the final field,
// this will allow us to steal the whole object.
function Object() {
  this.email setter = captureObject;
}

// Send the captured object back to the attacker's Web site
function captureObject(x) {
  var objString = "";
  for (fld in this) {
    objString += fld + ": " + this[fld] + ", ";
  }
  objString += "email: " + x;
  var req = new XMLHttpRequest();
  req.open("GET", "http://attacker.com?obj=" +
    escape(objString), true);
  req.send(null);
}
</script>
```

```
<!-- Use a script tag to bring in victim's data -->
<script src="http://www.example.com/object.json"></script>
```

The malicious code uses a script tag to include the JSON object in the current page. The Web browser will send up the appropriate session cookie with the request. In other words, this request will be handled just as though it had originated from the legitimate application.

When the JSON array arrives on the client, it will be evaluated in the context of the malicious page. In order to witness the evaluation of the JSON, the malicious page has redefined the

JavaScript function used to create new objects. In this way, the malicious code has inserted 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. Other attacks might override the default constructor for arrays instead (Grossman's GMail exploit took this approach.)

Applications that are built to be used in a mashup sometimes invoke a callback function at the end of each JavaScript message. The callback function is meant to be defined by another application in the mashup. A callback function makes a JavaScript Hijacking attack a trivial affair—all the attacker has to do is define the function. An application can be mashup-friendly or it can be secure, but it cannot be both.

If the user is not logged into the vulnerable site, the attacker can compensate by asking the user to log in and then displaying the legitimate login page for the application. This is not a phishing attack—the attacker does not gain access to the user's credentials—so anti-phishing countermeasures will not be able to defeat the attack.

More complex attacks could make a series of requests to the application by using JavaScript to dynamically generate script tags. This same technique is sometimes used to create application mashups. The only difference is that, in this mashup scenario, one of the applications involved is malicious.

### **3. Defending Against JavaScript Hijacking**

First-generation Web applications are not vulnerable to JavaScript Hijacking, because they typically transmit data as part of HTML documents, not as pure JavaScript. Applications that have no secrets to keep from attackers are also trivially safe from JavaScript Hijacking attacks.

If a Web application contains an exploitable cross-site scripting vulnerability, it cannot defeat data stealing attacks such as JavaScript Hijacking, because cross-site scripting allows an attacker to run JavaScript as though it originated from the application's domain. The contrapositive does not hold—if a Web application does not contain any cross-site scripting vulnerabilities, it is not necessarily safe from JavaScript Hijacking.

For Web 2.0 applications that handle confidential data, there are two fundamental ways to defend against JavaScript Hijacking:

- Decline malicious requests
- Prevent direct execution of the JavaScript response

The best way to defend against JavaScript Hijacking is to do adopt both defensive tactics.

#### **Declining Malicious Requests**

From the server's perspective, a JavaScript Hijacking attack looks like an attempt at cross-site request forgery, and defenses against cross-site request forgery will also defeat JavaScript Hijacking attacks.

In order to make it easy to detect malicious requests, every request should include a parameter that is hard for an attacker to guess. One approach is to add the session cookie to the request as a parameter. When the server receives such a request, it can check to be certain the session cookie matches the value in the request parameter. Malicious code does not have access to the session cookie (cookies are also subject to the Same Origin Policy), so there is no easy way for the attacker to craft a request that will pass this test. A different secret can also be used in place of the

session cookie. As long as the secret is hard to guess and appears in a context that is accessible to the legitimate application and not accessible from a different domain, it will prevent an attacker from making a valid request.

Some frameworks run only on the client side. In other words, they are written entirely in JavaScript and have no knowledge about the workings of the server. This implies that they do not know the name of the session cookie. Even without knowing the name of the session cookie, they can participate in a cookie-based defense by adding all of the cookies to each request to the server. The following JavaScript fragment outlines this "blind client" strategy:

```
var httpRequest = new XMLHttpRequest();
...
var cookies="cookies="+escape(document.cookie);
http_request.open('POST', url, true);
httpRequest.send(cookies);
```

The server could also check the HTTP `referer` header in order to make sure the request has originated from the legitimate application and not from a malicious application. Historically speaking, the `referer` header has not been reliable, so we do not recommend using it as the basis for any security mechanisms.

A server can mount a defense against JavaScript Hijacking by responding to only HTTP POST requests and not responding to HTTP GET requests. This is a defensive technique because the `<script>` tag always uses GET to load JavaScript from external sources. This defense is also error-prone. The use of GET for better performance is encouraged by Web application experts from Sun<sup>5</sup> and elsewhere. Even frameworks that use POST requests internally, such as **GWT**, document the steps necessary to support GET requests without mentioning any potential security ramifications.<sup>6</sup> This missing connection between the choice of HTTP methods and security means that, at some point, a programmer may mistake this lack of functionality for an oversight rather than a security precaution and modify the application to respond to GET requests.

## Preventing Direct Execution of the Response

In order to make it impossible for a malicious site to execute a response that includes JavaScript, the legitimate client application can take advantage of the fact that it is allowed to modify the data it receives before executing it, while a malicious application can only execute it using a `<script>` tag. When the server serializes an object, it should include a prefix (and potentially a suffix) that makes it impossible to execute the JavaScript using a `<script>` tag. The legitimate client application can remove this extraneous data before running the JavaScript. There are many possible implementations of this approach. We will outline two.

First, the server could prefix each message with the statement:

```
while(1);
```

---

<sup>5</sup> [https://blueprints.dev.java.net/ajax-faq.html#get\\_or\\_post](https://blueprints.dev.java.net/ajax-faq.html#get_or_post)

<sup>6</sup> <http://code.google.com/Webtoolkit/documentation/com.google.gwt.http.client.html>

Unless the client removes this prefix, evaluating the message will send the JavaScript interpreter into an infinite loop. This is the technique Google used to fix the vulnerability identified by Grossman. The client searches for and removes the prefix like this:

```
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json",true);
req.onreadystatechange = function () {
    if (req.readyState == 4) {
        var txt = req.responseText;
        if (txt.substr(0,9) == "while(1);") {
            txt = txt.substring(10);
        }
        object = eval("(" + txt + ")");
        req = null;
    }
};
req.send(null);
```

Second, the server can include comment characters around the JavaScript that have to be removed before the JavaScript is sent to `eval()`. The following JSON object has been enclosed in a block comment:

```
/*
[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
  "purchases":60000.00, "email":"brian@fortifysoftware.com" }
]
*/
```

The client can search for and remove the comment characters like this:

```
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json",true);
req.onreadystatechange = function () {
    if (req.readyState == 4) {
        var txt = req.responseText;
        if (txt.substr(0,2) == "/*") {
            txt = txt.substring(2, txt.length - 2);
        }
        object = eval("(" + txt + ")");
        req = null;
    }
};
req.send(null);
```

Any malicious site that retrieves the sensitive JavaScript via a `<script>` tag will not gain access to the data it contains.

## 4. Vulnerable Frameworks

A recent survey conducted by Ajaxian.com identified the 12 most popular Ajax frameworks in use today.<sup>7</sup> The survey results are based on the answers provided by 865 participants over the course of one week. Although the survey is not a scientific experiment, it gave us a starting point for understanding which frameworks are widely used.

One quarter of survey participants report that they use no framework at all. For this reason, JavaScript Hijacking cannot be addressed simply by fixing the popular frameworks. Web developers need to understand the risks involved in using JavaScript as a data transport mechanism so that they can protect the code they write and vet the 3<sup>rd</sup> party components they use.

When we began to analyze the frameworks, we quickly found that they can be divided into two major groups. Many provide client-side JavaScript libraries for implementing UI controls, but do not include a server-side component for end-to-end communication. In fact, only four frameworks from the survey list—**DWR**, **Microsoft Atlas**, **xajax**, and **GWT**—provide both client-side and server-side libraries for building Web applications. This distinction is important for determining how vulnerable these frameworks are to JavaScript Hijacking and the kinds of countermeasures that the frameworks should include.

The **Prototype** framework uses JSON as one of its primary data formats for communicating with the server. Because Prototype is a client-side JavaScript library, it cannot prevent JavaScript Hijacking on the server side by checking the validity of requests. However, it should give the server-side code the option of protecting the JavaScript it provides by accepting invalid JavaScript that cannot be `eval()`-ed without first being modified by the client. No such protections exist in Prototype, which leads programmers to create vulnerable applications based on the framework. The same is true for most other client-side Ajax frameworks, including **Script.aculo.us**, **Dojo**, **Moo.fx**, **jQuery**, **Yahoo! UI** and **MochiKit**.

All released versions of **DWR**, which means that versions up to and including 1.1.4, are vulnerable to JavaScript Hijacking. Until now, the framework has not built any mechanisms for preventing the vulnerability. The good news is that **DWR 2.0**—the version that is currently under development—is protected against JavaScript Hijacking by a mechanism designed to prevent cross-site request forgery. The protection leverages the fact that malicious script cannot read secrets stored in cookies set by other domains, which allows the framework to use a value stored in a cookie as a secret shared between the client and server. **DWR 2.0** automatically appends the session cookie to the request in the client and verifies on the server that each request contains the correct value. The **DWR** team is also doing preliminary work to add a mechanism that will prevent direct execution of the response.

**GWT** and **Microsoft Atlas** also use JSON to transfer data between the server and the client. By developing simple applications that were built using these frameworks and intercepting request and response traffic, we verified that both frameworks produce responses comprised of valid JavaScript that can be evaluated using a `<script>` tag and are therefore vulnerable to JavaScript Hijacking. By default, both frameworks use the POST method to submit requests, which makes it difficult to generate a request from a malicious `<script>` tag (since `<script>` tags only generate GET requests). However, as we mentioned earlier, both **GWT** and **Microsoft**

---

<sup>7</sup> <http://ajaxian.com/index.php?s=survey+popular+ajax+frameworks>

**Atlas** provide mechanisms for using GET requests. In fact, many experts encourage programmers to use GET requests in order to leverage browser caching and improve performance.<sup>8</sup>

**Rico** and **xajax** use XML to transfer data between the client and server and do not currently support JSON, which makes them trivially invulnerable to JavaScript Hijacking. However, both frameworks plan on adding support for JSON in future versions. Hopefully developers contributing to **Rico** and **xajax** will implement JSON support securely with the first version.

The results of our findings are summarized in Table 1 below.

Framework	Summary	Prevents JavaScript Hijacking?
<b>Dojo</b>	Supports JSON. Defaults to POST, but does not explicitly prevent JavaScript Hijacking.	No*
<b>DWR 1.1.4</b>	Uses an expanded version of JSON. <sup>9</sup> Does not implement any JavaScript Hijacking prevention mechanisms.	No
<b>DWR 2.0</b>	Uses an expanded version of JSON. Uses double-cookie submission to prevent XSRF and a <code>throw</code> statement in JavaScript responses to prevent JavaScript Hijacking.	Yes
<b>GWT</b>	Supports JSON. Uses POST by default; however, documentation describes how to make GET requests instead and does not mention any security ramifications.	No*
<b>jQuery</b>	Supports JSON. Defaults to GET.	No
<b>Microsoft Atlas</b>	Supports JSON. Uses POST by default, but allows programmers to easily change POST to GET and encourages doing so for performance and caching.	No**
<b>MochiKit</b>	Supports JSON. Defaults to GET.	No
<b>Moo.fx</b>	Supports JSON. Defaults to POST, but can easily be configured to use GET.	No
<b>Prototype</b>	Supports JSON. Defaults to POST when no method is specified, but is easily customizable for using either POST or GET.	No*
<b>Rico</b>	Does not currently support JSON, but will in the future. Supports XML as a data transfer format. Defaults to GET.	N/A
<b>Script.aculo.us</b>	Supports JSON. Provides additional UI controls and uses the <b>Prototype</b> library for generating requests.	No*
<b>xajax</b>	Does not currently support JSON, but will in the future. Supports XML as a data transfer format.	N/A
<b>Yahoo! UI</b>	Supports JSON. Responds to GET requests.	No

**Table 1. Analysis of Ajax Frameworks with Respect to JavaScript Hijacking.**

<sup>8</sup> <http://www.codeproject.com/Ajax/aspnetajaxtips.asp>

<sup>9</sup> DWR's format expands on JSON syntax to cope with recursive data structures and DOM tree parsing.

\* Indicates frameworks that reviewed early drafts of this report and plan to add a defense against JavaScript Hijacking in an upcoming release.

\*\* Indicates frameworks that are in the process of investigating the issue.

## **5. Conclusion and Recommendations**

The JavaScript implementations contained in popular Web browsers have been the source of numerous security problems, so JavaScript Hijacking follows a distinguished list of older attacks, including cross-site scripting and cross-site request forgery. (Bugs in JavaScript implementations have also led to no shortage of browser vulnerabilities.) The fundamental issue that allows JavaScript Hijacking, an eccentricity in the Web browser Same Origin Policy, is also used by benevolent programmers to create legitimate advertisements and application mashups, so it is unlikely that the Same Origin Policy will change in the near future.

The solutions and best practices proposed in this paper have been cobbled together in much the same way that rich web interfaces and mashup concepts themselves have been cobbled together. The Web programming community has stretched the existing technology well past its original purposes, so it is not surprising that we occasionally encounter an unexpected side-effect. These workarounds and hacks will be necessary until popular web browsers support standards that allow for things like secure cross domain requests and JavaScript sandboxing. Until then, we need to make programmers aware of the risks inherent in communicating confidential data via JavaScript.

To that end, we recommend that all programs that communicate using JavaScript take the following defensive measures:

- Include a hard-to-guess identifier, such as the session identifier, as part of each request that will return JavaScript. This defeats cross-site request forgery attacks by allowing the server to validate the origin of the request.
- Include characters in the response that prevent it from being successfully handed off to a JavaScript interpreter without modification. This prevents an attacker from using a `<script>` tag to witness the execution of the JavaScript.

We need to encourage Web application frameworks to be secure by default. For server-integrated toolkits such as **DWR**, **Microsoft Atlas**, **xajax**, and **GWT**, this is a matter of changing both the server-side and client-side components. Purely client-side libraries cannot defend against JavaScript Hijacking without support from server-side code, but they can still improve security by making the JavaScript Hijacking risk clear to developers and by including features that make it easy to implement secure server-side code.

If programmers want to create applications that can participate as part of a mashup, they should be required to explicitly disable a security constraint, and the framework should make them aware of the consequences of their actions.

## **Acknowledgements**

We would like to extend special thanks to a number of people for their feedback on early versions of this paper. We thank Jeremiah Grossman for his valuable comments on the technical content of the report, Joe Walker – for a great discussion of DWR security, Alex Russell – for a discussion of Dojo framework and highlighting important points missing from the original draft, and Bob Ippolito – for a discussion of MochiKit framework and suggestions on how to address the issue. Eddie Lee, Adam Murray, and Erik Cabetas helped us proofread the draft and provided insight into their experience with Ajax and Ajax frameworks.