A **Cross-site Request Forgery,** aka **CSRF** or **one-click attack**, is a diffused security issue issue where unathorized commands are sent from the user's browser to a web site or a web application. CSRF is different from[Cross-Site Scripting](http://en.wikipedia.org/wiki/Cross-site_scripting) in the sense that it does not need to inject code into trusted pages, but can work from untrusted ones thanks to the open architecture of the web.

**A quick example**

I was browsing the bug tracker of a popular web service to propose an improvement, where I saw that the only *Critical* ticket was related to CSRF vulnerabilities.

The website uses links for the deletion of the current user's data, such as /delete\_my\_items. When a user follows the link present on his account page, his session cookie establish his identity and his data is deleted.

To perform an attack, we only have to make the user load this URL. A simple link would probably not work, but we can easily reach the same goal with an image:

<img src="http://application.com/delete\_my\_items" />

We just have to provide a link to a page containing this image for the user, or send it to him in an HTML email.

**How it works**

Unlike for XMLHttpRequest, images and other resources can be loaded from whatever hostname, and even if the image tag is hosted on a different website it won't matter.

Since the request is performed by the user's browser, it will transmit the session cookie as it would for any application.com page.

The point of all CSRF attacks is **tricking the browser into sending an *unwanted* HTTP request**, so that there is no need to steal the identity of the user. That's probably why our banks log us out after 2 minutes of inactivity, or require one-time tokens to authorize money transfers.

**The assumptions of CSRF**

In order for a CSRF attack to be possible, some assumptions have to be verified:

* the attacked website **does not check the Referer** HTTP header, so that it accepts requests originating from external pages.
* The web site **accepts data modification via form submissions or URLs** that have side effects which the attacker can exploit.
* The attacker can **determine all the values for the request inputs**. In the simplest case, authentication is done exclusively via a session cookie and so the attacker just have to fill non-sensitive fields.
* The user must **load a malicious page** containing the attacker's code. Judging by the amount of Facebook Likejacking, clicking on everything that moves is a pretty common behavior.

**Countermeasures**

Every countermeasure can raise the bar and make CSRF more difficult to perform.

First of all, **don't use GET** for operations with side-effects. CSRF relies on requests that produce side-effects like deletions or data modifications, and using GET just makes these requests simpler to perform.

**POST however is not enough**: an attacker can submit a form using JavaScript once the page is loaded, and create a phantom POST request.

There are different alternatives to avoid CSRF for POST requests.

**1. The Referer header**

You can **check the Referer HTTP header** and verify that the request originated from a page internal to your web application. The Referred can be spoofed very easily by a malicious user agent, but the common user is running a general purpose browser, not a malicious user agent.

This approach assumes no one can inject HTML/JavaScript in your pages to originate a request; otherwise the referrer would be correct.

The issues is that **referrer stripping is really common**: [up to 11% of HTTP requests do not contain it](http://www.adambarth.com/papers/2008/barth-jackson-mitchell-b.pdf); for example, corporate proxies or old user agents commonly strip it away. Thus, strict referrer checking would lock out more than a tenth of your user base from making POST requests.

**2. Session token**

Solutions based on a **one-time token for forms** are by far more popular: the token is saved in the user's session and transmitted in each official form.

[Zend\_Form\_Element\_Hash](http://stackoverflow.com/questions/1627732/how-to-use-zend-form-element-hash) is an example of this technique: an **hidden field** is added to forms and **populated with the token**. To make a successful request, the attacker must know the user-specific token, but can't access it since an external page cannot load a form on another domain via JavaScript.

In 2011, this is *the* standard approach.

**3. Double-submitted cookie**

A variant of the previous technique consists in **matching a token sent with the form *and* with a cookie**, instead that with a session value.

Due to the [same origin policy](http://en.wikipedia.org/wiki/Same_origin_policy), JavaScript code loaded from another host cannot read a cookie set on the official hostname. Thus while official forms will send the value along with the cookie, the attacker's ones will have to guess the value of the cookie to include it in the request (an impossible task if the value is long enough.)

The advantage of this variant is that less server resources are occupied to store form tokens, and you won't be the subject of a DDOS attack based on multiple form loading.

**Conclusion**

The proposal of the most famous paper on CSRF is to add an Origin header to HTTP, in alternative to Referer. This head would be sent only for POST requests, and would contain only an hostname instead of a full URL, to ensure privacy.

The techniques described in this article are viable and worth a thought for any application that contains useful data; they hinder testability with certain instruments that build HTTP requests, but they are not an issue for browser-based tools like Selenium