Cross-Site Request Forgery
Mikko Rantanen
University of Helsinki
Gustaf Hällströminkatu 2b
mcrant@cs.helsinki.fi

ABSTRACT
Cross-site request forgery (CSRF) is a web site vulnerability, where a user and her browser can be tricked into sending unsolicited commands or data to a vulnerable web site. A malicious site can use Javascript to trick victim browser to POST form data to vulnerable site or do a GET with unwanted side effects. Successful exploitation does not necessarily require the user to be logged in to a website, as the vulnerability allows attacker to cause log in using credentials known to the attacker without user intervention (so-called 'login CSRF' variation). Unsolicited commands could be for example monetary transactions to victim's bank web interface or reconfiguration commands to victim's home DSL/cable router. Login CSRF can be used to trick the user into thinking she can safely input data, such as credit card numbers, into her own account, when in reality the information is leaked to the attacker. In this paper, we describe CSRF, its history and protecting against it in detail, and explain currently available state-of-the-art solutions.

Categories and Subject Descriptors
D.2.0 [Software engineering]: General – protection mechanisms, standards.

General Terms
Security.

Keywords
Cross-site request forgery, CSRF, XSRF, confused deputy, web security, same origin policy.

1. INTRODUCTION
Web-based services use HTTP and HTTP/S as their application layer protocols to exchange data with clients. The protocols are stateless, and therefore the web site and user agent (the browser) must keep and maintain state information themselves. In addition to plain HTML, client and server exchange metadata in the form of headers. When a user logs in to a web site, he or she often provides some credentials, such as user name and password, using HTTP POST method. Usually the web site generates one or more cookies, which can be any string, often a random string or nonce, and the browser stores the cookie. The cookie (or cookies) can then be considered a shared secret and used by the browser as a proof of identity and as a session identifier towards the web site.

In general, browsers comply with so-called same origin policy (SOP) [30], which means that whatever credentials were received, they will only be sent back to where they originated from and access to them allowed only to HTML/Javascript originating from the sending site. Otherwise, a third party could steal a valid cookie for some user's web banking session, for example.

As long as browsers properly enforce SOP and take care that any supplied Javascript can't see other domains' or documents' cookies, the credentials should be safe. But SOP only solves the problem of keeping credentials safe, not who can use them. Any site that redirects a browser to another site, from where the browser has stored cookies, makes the browser to send the credentials and the receiving site does not necessarily have reliable way to know where the request or redirect really originated from. If the application (site) is not properly checking or trying to check where the request really came from, it can be exploited by planting a specially crafted URL into any other site. This is called CSRF vulnerability and it is a flaw in the web application, in the site, that receives the credentials or cookies from the browser. In addition to this, it is possible to use Javascript to programmatically force a browser to log in (submit POST form) to the site using credentials known to the attacker. All web applications should check that the user or browser has not been tricked into the site from a third party site. This is best accomplished using some combination of secret tokens, cookies and headers, or programmatic methods such as XMLHttpRequest.

The rest of the paper is organized as follows: in section 2, HTTP, its working and associated terminology is reviewed; in section 3, CSRF, browser actions and related issues are explained; in section 4, threat models and currently known and used countermeasures are detailed; in section 5, four historical CSRF incidents are presented. Section 6 summarizes.

2. About HTTP
HTTP protocol comes in two versions, 1.0 and 1.1, of which 1.1 is almost exclusively used. When client and server communicate, the client uses HTTP methods (such as GET and POST) and server replies with status codes, such as 200 OK, plus the actual content (HTML, Javascript, XML etc.) which is called message body. [25]

When talking about Javascript, the scripting language dialect of ISO/IEC 16262 compliant ECMAScript [45] is meant.

2.1 Headers
In addition to the HTTP methods like GET and POST, HTTP uses headers to communicate metadata related to the connection. The headers are seen and interpreted by the client, server and any intervening proxy/cache devices.

Standard headers relevant to CSRF are Referer, Origin, Cookie, and Set-Cookie. Additionally, cross-origin resource sharing (CORS) specifies total of nine (9) headers [29] for controlled cross-site requests, including the Origin header proposed in [6]
and defined in [34]. Web site designers can also create custom headers programatically using XMLHttpRequest-method [32].

In particular, the protected headers Referer or Origin [32] tell the receiving web site from what site or URL the browser came from.

### 2.1.1 Referer
Referer-header is sent by the browser and contains the complete URL of the page where user came from (programmatically or by clicking a link). It is sometimes suppressed by firewalls, privacy protection software or browser policy, as it may contain private information about who and when browsed what page, or in the worst case, leak user credentials.

### 2.1.2 Origin
Origin header is similar to Referer header, but it only contains the URL up to the site-part. The specific page information is left out. This header is defined in both CORS candidate recommendation [29] and RFC6454 [34] section 7.

### 2.1.3 CORS
Controlled cross-origin resource sharing is available in modern browsers. In practice, it's nothing else than out-of-band data exchange between browser and server using HTTP headers. The server can specify to the browser from where (and with what constraints) cross-origin requests are allowed and browser sends Origin-header, which is the key part in CORS. From browser and Javascript perspective, Internet Explorer is a little bit different from the others; this is discussed in [35, 36], but [37] claims that CORS headers work in 'normal' manner since Internet Explorer 7.

### 2.1.4 Limitations
As described in [32] section 4.6.2, particular headers may not be set programatically. They are considered part of the browser-enforced security model to disallow, for example, forging Referer- or Origin-header.

### 2.2 State
Statelessness and state, in context of HTTP protocol, mean that each and every GET or POST request made by client (browser) towards the server, are separate, without any connection between them. Because of this, state must be created and maintained somehow if such a connection or association between user's any two clicks must be made. A stereotypical example is a shopping cart, where server must identify the browser and associate it with the correct cart. This is traditionally done using cookies [28] and maps loosely to OSI model session layer [27]. It is possible to encode unique ID into the URL or a unique field into a form, and use both cookie and this unique ID together.

### 2.3 Example with GET / POST
HTTP protocol methods GET and POST form the vast majority of all browser requests. In GET method, the browser asks for a particular URL and then tells from what host it is looking from. In POST method, the browser submits data to some specified host for processing. The data is name-value pairs. In both types of requests, the browser automatically submits any cookies it has for the host it is contacting.

In both cases, it is possible for the browser to send referer information, which tells exactly from what URL the browser was coming when it made this GET/POST request. In the following examples, GET request does not contain Referer- or Cookie-header but POST request does.

#### 2.3.1 Example GET
GET /login.html HTTP/1.1
Host: rv.tko-aly.fi:80

#### 2.3.2 Example POST
POST /userauth.php HTTP/1.1
Referer: http://rv.tko-aly.fi/login.html
Content-type: application/x-www-form-urlencoded
Host: rv.tko-aly.fi
Content-Length: 42
Cookie: RV=8pourts602vuk5uij3ljql56
username=nonexistent&password=LIY/D40f&log=1

### 2.3.3 Parameters in GET
It is possible to create a GET request that transmits some parameters to the site before there is any existing session between the user's browser and the site. For example, appending string ';charset=UTF-8' to GET request of http://www.google.com/, the Google web site will serve the page in English and not redirect your browser to localized size such as www.google.fr.

In this manner, it is possible for the server side to create unique URLs for GET and use them to transmit, for example, unique session ID. This is strongly discouraged, at least as the primary session ID mechanism, as the URL is often stored in log files and available to various proxy/cache devices whereas cookies are not.

### 2.4 Cookies
When a server wishes to maintain state with a client (almost always), it creates a cookie (unique and typically random string) and sends it to the client. In the previous Example GET, the server reply (simplified) might be along the lines

HTTP/1.1 200 OK
Host: rv.tko-aly.fi
Set-Cookie: RV=8pourts602vuk5uij3ljql56; expires=Thu, 28-Feb-2013 12:34:56 GMT; path=/
Content-type: text/html; charset=UTF-8

Here the server sets a cookie named 'RV' with some random value of numbers and letters, containing about 128 bits of randomness. This cookie has two attributes, Expires and Path, set as well. The possible attributes are explained in [28] section 4.1.2, but interesting attributes limiting cookie exposure are Domain, Path, Secure and HttpOnly:

- Domain attribute controls to what less-specific hosts the browser may send the cookie
- Path controls which paths in allowed hosts may receive the cookie
- Secure mandates that the HTTP connection must be over secure transmission (like HTTPS)
- HttpOnly avoids the cookie to real HTTP requests only (denies access to the cookies from scripts, from browser API etc.).

### 2.5 XHR / XDR
XMLHttpRequest (XHR) is the ‘standard’ method name for programmatically creating a HTTP request [32]. It enables the developer to make a HTTP request to any target and specify custom headers, but does not allow setting Referer, Origin or such protected headers. It is possible to build a site to extensively use XHR and send custom headers in every request.

XDomainRequest (XDR) is Microsoft’s name for creating a CORS [29] request or cross-domain AJAX request, as Microsoft calls it (see XDomainRequest under [42]). It follows the W3C draft [29] and has been available since Internet Explorer 8.
3. CSRF and associated threats
Cross-site request forgery vulnerability is a variation of the so-called confused deputy problem. In the confused deputy problem, a computer program, web application or similar, is fooled by a user or an attacker into performing an action using capabilities and privileges assigned to the program itself, not those assigned to the user or attacker. A good example of this problem is described in [1], but undoubtedly earlier examples exist.

3.1 Regular CSRF
In a regular CSRF vulnerability, the attacker assumes the user has logged in to some vulnerable web service which the attacker wishes to exploit. The victim's browser has at its disposal a cookie or credentials associated with the site that is being exploited. If the browser can be manipulated to send a GET or POST request to the victim site, it will also send the cookies associated with that site automatically. Manipulation can occur through user interaction like clickjacking [2], script injected to some forum site or similar XSS [5] treachery, or just an `<img>` tag in a forum post [19]. This way the attacker indirectly uses the victim user browser's state to his advantage.

3.2 Login CSRF
In another type of vulnerability, dubbed login CSRF [6], an attacker uses credentials known to him and causes victim user browser to log in to a vulnerable site using credentials known to the attacker. At first, this may sound counterintuitive, but if the site in question is a search engine, the attacker can then log into his account and see everything the victim has searched or otherwise interacted with the site. Naturally, this applies to sites

![Figure 1: Traditional CSRF attack. [38]](image1)

![Figure 2: Event trace diagram for a login CSRF attack. [6]](image2)
like PayPal, where victims could be lured into revealing sensitive financial data. An example of using known default credentials and subvert victim user's DSL modem is presented in chapter 5.4.

3.3 Other related issues and vulnerabilities

Conditions leading to a CSRF exploitation do not necessarily need exploiting any other typical vulnerability exploits, but they can be used to assist it.

3.3.1 Overwriting cookies

As mentioned earlier, SOP ensures the scripts can only access methods and properties that share the same origin. The SOP applies to cookies as well, so a cookie is only available to a site or script from a site that originated the cookie. Unfortunately, there are additional problems with properties associated with cookies.

A fraudulent site can, for example, overwrite cookies of an honest site under certain conditions due to cookie scope, or create identically named cookie that has the same scope as the cookie of an honest site [3, 31]. Worse yet, an active network attacker can insert a Set-Cookie header to unencrypted HTTP connection and overwrite a secure cookie [31]. This is because the cookies' Secure-parameter does not provide integrity protection and the receiving site does not receive information about how the cookie was set.

This enables the attacker to create conditions similar to login CSRF attack. In this case, user is given a cookie that is associated with attacker's credentials by a vulnerable site.

3.3.2 Forging headers

In the past, there have been vulnerabilities in browsers allowing forging of at least Referer-header, but this problem has not resurfaced [6].

If Referer-header is used by the site to verify the origin where the browser is coming from, this leads to a regular CSRF condition, where site thinks the user is knowingly interacting with the site. This would work in the same manner for any header, including custom headers (programmatically from XMLHttpRequest) and those used by CORS.

3.3.3 Clickjacking

While not a CSRF per se, we'll mention clickjacking here to distinguish it from CSRF.

In clickjacking, the IFRAME tag can be used to create invisible frame of victim site on top of attacker's site. The invisible frame can be positioned so, that buttons or links in the victim page are on top of attacker's page where victim user is likely to click. This way the attacker hijacks user's clicks to do attacker's bidding [2, 33].

The important distinction between CSRF and clickjacking is, that in clickjacking, the user interacts with UI elements of the target site but sees and thinks he's interacting with some other site. In CSRF, no interaction on user part is required.

3.3.4 Cookieless authentication schemes

If cookies are disabled in the browser, it is still possible to use parameters or anchor tag in GET to transmit a session variable. This is discouraged for (at least) following reasons:

1. Session fixation. An attacker can visit or login in to a page without cookies and obtain valid session ID, then instruct victim browser to use this URL for victim's GET. Now victim is using a site as the attacker.
2. Copy-pasting URLs or taking screenshots contains a valid session ID. For example, sometimes URLs are copy-pasted to a public forum.
3. The session ID is visible to intervening proxy/cache devices.
4. The session ID is stored in log files, including browser history.

Logically, as all elements are under attacker's control, defending against CSRF is not possible in cookieless authentication scheme, even with secret token in POST form.

3.3.5 Origin, content type and authority

A summary on subject of authority granted by browsers to various content-type (header set by HTTP server) is given in [34] sections 3.3 and 3.3.1. In short, content-type such as text/html has full authority to its origin, whereas media like image/png has no access to resources or objects in its origin. For example, when a user writes to a forum in some web site, the web site receives content (input) from this user. This user may or may not be malicious and may or may not be validated. The web site should treat the user-entered data as suspicious and assign different content-type than text/html to try and avoid browsers interpreting user-entered content as code. Unfortunately, as [34] notes, many browsers try to guess the content-type based on the actual stream of bytes from the server instead of using server reported content-type. This may lead to a situation where content without authority is granted full authority.

3.3.6 DNS Rebinding

An attacker can buy a domain name and associated DNS services for a few dollars per year. When the attacker controls the forward records (A or AAAA), a domain name www.evil.com can be made to point to any IP address, including private IP addresses (RFC1918), non-routed IP addresses or other normally not reachable from Internet. When a victim is lured to the attacker's domain the victim will first get real IP address and the attacker can serve the victim malicious Javascript. The piece of code then makes another DNS query for same name, but receives different reply. Attacker's Javascript has now access to the IP address attacker served the second time. [39]

3.4 What can be had and how

Given that links and following them is the fundamental idea behind WWW, the browsers allow web sites to provide links anywhere. This policy lets anyone control other users' browser(s) and redirect them to sites specified by the attacker. This is potentially dangerous, because victim's browser has access to:

- Network connectivity within victim domain. Even if victim sits behind a firewall, victim's browser is inside the perimeter guarded by the firewall. Even services running on otherwise unreachable localhost are available to the attacker.
- The browser state of the victim. Attacker can read (and use) victim browser's state, because the browsers send their credentials and cookies automatically when accessing sites. Attacker can also write part of browser state by redirecting victim browser to a site that sends a cookie, causing victim browser store this cookie (see 3.3.1 for example).[6]

Based on previous chapters, it should be evident that any site can contain potential CSRF vulnerability exploits:

- Most people allow Javascript without any restrictions, so unless they selectively enable Javascript for sites or
use NoScript [52] or similar plugin, their browsers can be easily targeted. This can happen via a media where content is usually restricted, such as a forum posting (free) or ad network (cheap).

- If the attacker has complete control of the web site content, the attacker is exempted from any restrictions imposed by forum software, ad networks or similar. A VPS or just web hosting can be had for few dollars per month and users can be lured in via, for example, advertisements.

- Additionally, if the attacker can mount an active network attack, he might gain control of DNS the victim uses. HTTPS is designed to combat this type of attacks.

As a summary, the root of the problem is that browsers are not necessarily able to tell the web site/application, from what site they came. Even if the Referer-header is available, vulnerable web sites don't pay attention to this or otherwise verify the intention of the victim.

4. COUNTERMEASURES

We list here countermeasures that may be used to offer protection against CSRF attacks. In addition to those evaluated in [6] and still valid today, a Javascript-based transparent solution is presented.

4.1 As proposed in [6]

Barth et al. list three defenses from late 2008 that are still usable and valid, if implemented correctly:

1. Validating secret token in forms
2. Validating Referer-header
3. Custom headers using XMLHttpRequest

All of these schemes do have some drawbacks despite their utility.

4.1.1 Token validation

Using secret tokens in forms is effective against both regular and login CSRF, but developers may forget to use it because they must create a pre-login session when authenticating users. The selection of secret token is also open to debate: should it be session independent or dependent? Independent tokens require the server to keep state, but dependent tokens can be cryptographically strong tokens, based on session ID known to server only. If a common secret can be shared among the site servers, an HMAC or encryption based token can be created without the need to maintain large state table. Sharing just the secret, not state, is proposed for example in TCP Fast Open paper [41].

The site must also verify that the request contains a cookie tied to the session with the generated secret nonce, so this bijection must also be checked using HMAC or other mechanisms (such as described in [43]) to avoid keeping state.

If there is existing code that requires fixing, it can be costly to find all the places where CSRF protection is required and add the functionality to existing web application. While effective and usable, this technique really requires support, preferably automatic, from the framework used to create the application, or some external device or code.

4.1.2 Referer header validation

Two possible styles of any HTTP header validation can be identified:

- **strict checking**, where the header is required by the server to be present (and contain proper value) and
- **lenient checking**, where the server allows requests without the header.

Rudimentary protection against CSRF vulnerability is traditionally made using strict checking of the Referer-header and checking that the request comes from some expected location. When making a GET or POST requests, the browsers, by default, do send the Referer header. But it contains the full URL, including the path and parameters of the referring site, so it is sometimes suppressed [6, 40] by privacy extensions or personal firewalls. It might also be falsified using Flash [40], a browser bug [44] or some other software or a yet unknown vulnerability. It is not mandatory header in any way, after all.

In addition to this, an attacker can redirect victim's browser to ftp://attacker.com/index.html and then cause cross-site request. Because it is FTP URL, the browser does not send the header at all, and would therefore pass lenient checking. The header is unreliable by its nature even if it solves the problem in theory.

In practice, [6] measured the transmittal of Referer header using two ad networks and concluded that 3.11% of the times the Referer header was missing when using plain HTTP, so strict checking is not possible over plain HTTP as it would deny access to many legitimate users. Lenient checking is yet again subject to fail.

The main conclusion from [6] is that strict Referer header validation is a viable mechanism, if used over HTTPS, because according to their measurements, only about 1/5000 HTTPS requests suppress the header. The authors conclude that this is an acceptable level of refused service requests.

What is left unsaid is, that implementing this easy defense also allows for controlled cross-site origin requests to be made, if the exact cross-referral URLs or their scheme-, domain- and port-specific parts can be iterated and whitelisted in the web application. It is the authors opinion that this is a viable approach for completely in-house grown solutions if talking about sufficiently small environment.

4.1.3 Custom headers

Web application developers can use Javascript function XMLHttpRequest to add custom headers and create GET or POST request. Because of the way how browser-enforced SOP works, scripts can add custom headers only to requests going towards their origin, so the web application can rely just on the presence of a predefined custom header. As the script is controlled by the web application, it can be customized on the fly if needed. An often-quoted header is X-Requested-By: with variable content, but the name and associated data can be almost anything.

It is therefore possible to build the web application to rely solely on XMLHttpRequest-method to protect against CSRF - even NSA SNAP thinks so [46]. On the other hand, this is cumbersome and puts the burden of identifying the necessary places to the shoulders of developers. And where there is human, there is place for error.

4.1.4 Other suggestions

The team in [6] suggests new, Origin header instead of Referer header and claim the XMLHttpRequest working group and Microsoft adopted this naming convention. They also provided patch to WebKit, the Safari open source component and wrote extension to Mozilla. After looking into the matter, it seems...
the Origin header has not gained widespread use [4], possibly because it is now part of XMLHttpRequest specification and RFC6454. The team's work is laudable, though, and they mention (then still in the future) concept of cross-document messaging used by [4].

4.2 Past attempts

There have been multiple attempts in the past to mitigate the CSRF problem by creating external components to automatically fix vulnerable applications. Of those, worth mentioning are following three, two server-side and one client-side defense plus one particularly clever transparent solution (jCSRF):

1. CSRF Guard [53]. This is OWASP attempt at protecting TomCat applications, but [38] shows it is vulnerable to login CSRF and XSS. It also works only with TomCat and can't handle dynamically created pages [4, 6].
2. NoForge [47]. This is one of the earliest defenses against CSRF attacks. It does not support dynamically created pages [4, 6] and suffers from login CSRF vulnerability. It also leaks the CSRF token to external sites, as it does not discriminate where it sends the token [6, 47].
3. Firefox Extension [48]. Client-side protection using context and perceived user intention by interpreting HTML, works for GET requests too. The value of the work is diminished as it only works on old Firefox versions (3-series). Their work shows that about 47% of all browser requests are cross-domain.

Despite the apparent simplicity of the problem, there are many incorrect implementations that neglect some aspect of the problem.

Namely, most efforts seem to fail due to one or more of the following reasons:

- Human factors. Developers need to be constantly acutely aware of the risk and actively combat it. At least Django and Ruby on Rails have taken initiative to fix this, but there is always room for improvement [50]. In addition to this, many web sites use GET requests for state altering functions (discouraged by [25], section 9.1.2).
- Specific solutions. Patches to browsers and browser-specific plugins or extensions. Possibly incompatible with later browser versions and only available for small subset of browsers being used.
- Blocking legitimate cross-site requests. This problem is solved using CORS [29], which is partially supported in IE8 and 9, fully supported in IE10 [42, 49]. Most other current browsers support it as well [49].
- Neglecting to protect against login CSRF.

4.2.1 jCSRF

jCSRF [4] is an attempt to provide solution to aforementioned problems with the exception of not supporting GET requests. It is transparent to both server and client. As a concept it is promising, but suffers from the drawback that for full compatibility, it must be implemented as a separate device. Otherwise, it must a loadable module within the web server software. HTTPS must also be terminated in the external device, if modular software solution is not used.

jCSRF is a transparent proxy between server and client. It inserts one cookie and one line for client-side Javascript into HEAD section of the server-sent HTML reply pages. jCSRF expands on the double-submit technique described in [43] and this ensures that login CSRF is not possible. Double-submit means that a secret token in POST or a GET parameter must equal that of the cookie - hence the name 'double-submit', as the secret value is sent in both the HTTP headers (metadata) and the request itself. jCSRF cookies are authentication tokens, made by encrypting a nonce with a random key, and the nonce is created deterministically using a monotonically increasing counter from sufficiently large number space. The cookie can then be compared to the expected value seen in the GET and POST requests, thereby tying them to the cookie which is the authentication token.

The tricky part in implementing controlled cross-site origin requests is implemented using the postMessage API call introduced in [6]. As its name implies, the method is used to send a message but it also allows specifying that the recipient (target) frame, or document, must come from origin specified in the API call. Assuming we have a page S with intention to use resources from another site T, the jCSRF implementation does following:

- the source page (S), who wishes to make a cross-origin call, contains jCSRF code and knows the target (T).
- the jCSRF code hooks the onSubmit() method, realizes that cross-origin resources are requested when user presses 'Submit', so it gets an IFRAME from target (T).
- the IFRAME code from T proceeds to make XMLHttpRequest back to T, sends 'S' (identifier, hostname or similar) as content and gets a cookie by name of authentication token (C_u, a random nonce) and content of AES_k(C_u || S). The AES key is known only to T.
- the IFRAME (from T) uses the postMessage method and specifies that the framing (recipient) document must be from origin S.
- the browsers enforce SOP and allow the message only if the framing document of the IFRAME is of origin S; therefore, attacker can not trick T to send to the attacker.
- the framing page contains jCSRF code. It inserts the received token to POST form and passes it to T.
- T receives, decrypts and checks correctness of the token, associated with C_u, has created.

This technique has its pros and cons. The major drawbacks are that the solution requires an external box and must terminate HTTPS, if it is to be protected. The advantages are transparency to both clients and servers (clients must run Javascript, though), transparency to developers (removal of human factors, ease of deployment) and controlled cross-site requests.

4.3 Luddite approach

It is not possible to protect against this type of vulnerability solely on client side, because the flaw exists in the server side application. Even if Javascript is disabled and only GET can happen without user intervention (such as with <img> tab), it is still possible that the server side implementation causes side effects from GET that trigger unintended behaviour. As [6] recommends, different cases need different defenses.

Protecting against vulnerabilities described in chapter 5 might had been possible by keeping the vulnerable site open in a different browser (possibly in a virtual machine). Different browser here means a browser that does not share the credentials or session identifiers with any other window or tab of the same browser executable (e.g. Internet Explorer's InPrivate Browsing or similar), or different web browser software altogether.
This approach would require relatively big cognitive efforts on users' part, so it is not likely to gain acceptance. The inconvenient truth is, that humans tend to live in denial and often choose easiest route to achieving contentness, such as keeping Facebook and web mail constantly open while surfing less salubrious parts of Internet in other tabs.

Few of author's colleagues use snapshotting filesystems and run some operating systems in a virtual machine. After using the browser in this VM for a particular activity, they tell the FS to restore the previous snapshot. This also denies or severely limits (at least for the duration of running the VM) any malware that otherwise infect the OS in the VM.

4.4 CORS

Cross-origin resource sharing solves the problem of allowing legitimate cross-site requests. It does not address login CSRF concerns and puts some of the burden to the client side (Introduction, in [29]) as it defines the Origin header (can be considered replacement of Referer header). CORS has been in the works since 2005 [29], and the glacial pace of W3C has managed to push it into candidate recommendation status. It is implemented in all modern browsers to the extent that it's usable, so only thing that's left is for the web sites to deploy and configure it. A good tutorial is available in [51] for interested readers.

In short, CORS classifies HTTP requests into simple and non-simple ones. Simple ones are GET, POST and HEAD verbs with restricted subset of headers. For example, if Content-Type header is present, its value can only be one of application/x-www-form-urlencoded, multipart/form-data or text/plain. Simple cross-origin requests can be made by the browser at any time by just including the Origin header.

Non-simple requests use other verbs such as PUT or DELETE. Browsers must ask from the server, using OPTIONS verb, if the non-simple request is supported by the server before making the request.

When CORS-aware browser makes a cross-site request, it must always include the Origin header, that contains the scheme, host and port of the site where the browser came from (the web origin concept, [34]).

The web server, if it's CORS aware, includes at least Access-Control-Allow-Origin header in its reply that tells the browser if the cross-origin request was accepted or not. [29, 51]

5. HISTORY AND SOME EXPLOITS

Despite the problem having been known and well-defined since at least February 2000 [7], it has persistently stayed in Open Web Application Security Project (OWASP) top-10 list of most prevalent and dangerous web application vulnerabilities since 2007 [8, 9]. During the six elapsed years, it has only moved down three steps in the list from position A5 to A8 (this happened between 2010 and 2013 [13, 9]). Within OWASP organization, the nomenclature apparently did not include CSRF or XSRF yet in 2004, but MITRE data ranked it 5th year 2006 and OWASP "raw data" as 36th year 2007 [8].

In 2007 according to OWASP, and I quote verbatim, 'All web application frameworks are vulnerable to CSRF' [8]. This is fortunately not the case anymore as [9] notes. Looking at history of some big frameworks such as Django (widely used) and Tornado (used by Facebook), it can be seen that both have had difficulties providing adequate protection against CSRF [10, 11], but that's not all. Most, if not all, web frameworks, including as an example Google's webapp.py, require the programmer to explicitly think, code and protect against the CSRF attacks [12]. The case being that even big corporations, who should know the risks and how to mitigate them, fail to do so, I present here shortly four different cases from past six years demonstrating various successful exploits, the techniques used and their impacts.

5.1 December 2007, GMail

Google has had multiple CSRF vulnerabilities in the past [15, 16]. Of those, the one described in [16, 17] is of very much in spirit of CSRF: stealthy, evades detection easily, and requires vigilant,
lucky or paranoid user to be detected. The vulnerability allowed attacker to insert filters into a victim user's GMail account and as described in [14], caused the victim to lose control of his domain name. Of note here is that the victim wrote in his blog what he'd be doing and when he'd be leaving for a long trip, and the process of domain name transfer started on the exact date of his departure. This particular vulnerability allows the attacker to see all private e-mails of the victim and selectively block victim from seeing messages from a particular sender. By creating a filter to forward and then delete (without storing in victim's inbox) the messages, the victim was completely unaware that his domain name was being transferred.

This is an example of how relying on e-mail alone for authentication is a bad idea. A properly implemented two-factor authentication would had prevented the incident.

5.2 August 2009, Facebook
Facebook had similarly stealthily exploitable CSRF vulnerability at least until late 2009. This is described in [18], and caused Facebook to leak victim's profile picture, name and friend list (with their picture) to the attacker. The original post [19] remembers to mention that any added cross-site scripting (XSS) is enough to reveal victim's e-mail address as well.

As the description along with the demonstration video of the vulnerability show, there was a rich harvest to be had for spammers. The proof of concept (POC) demonstrated a popular forum site software that allowed the exploit via <img> tag, and users in a forum are usually interested in whatever topic the forum is specialized in. The exploitation would had taken place right under the victim's nose because the design and implementation of Facebook's automatic authentication (AA) allowed redirection to external sites and made the exploit invisible to the end user.

This vulnerability demonstrates exploitability without any Javascript, e.g. using a redirection causing a regular HTTP GET. While its impact was low, normal protection by disabling Javascript from unknown sites has no effect. Side effects from GET are discussed in [25], but it states that 'it is not possible to ensure that the server does not generate side-effects as a result of performing a GET request'.

5.3 January 2010, OpenCart
Early 2010, a hacker and blogger Ben Maynard contacted OpenCart developer after noticing CSRF vulnerability. Mr. Maynard explained in detail the vulnerability to the developer, who refused to understand that the problem is in his code, not in the client side. In less than two weeks, the problem reporter had forked the code and fixed two additional problems or vulnerabilities in the software. [21]

OpenCart still exists, is being actively developed and claims PCI compliancy, localization and integration to over 20 payment methods, including PayPal and Klarna [20]. This an example of a too big an ego and too little humility on developer's part, where the developer won't admit that his software suffers from a common flaw. In the end, it seems that the vulnerability got fixed, but it took some harsh words and probably a bruised ego. Additionally, this touches base with CS101 lessons about ethics and diligence.

5.4 2011, DSL modems
Late 2012, Kaspersky Lab expert Fabio Assolini wrote a summary story about what happened in Brazil. Beginning early 2011, DSL modems of six different hardware manufacturers were silently broken into en masse.

A total of about 4.5 million DSL modems were compromised using a simple Javascript-based CSRF vulnerability. The attackers changed the modems' access passwords and the DNS servers using about 40 malicious DNS servers under attackers' control. The attackers' DNS servers lead victims to false pages mimicking those of common Brazilian banking web sites. In some cases, the DNS servers tricked victims to other false pages hosting malware and allowed attackers to take control of victims' machines. [22]

CERT Brazil released a bulletin about the compromised DSL modems in March 2012, and Brazilian Ministry of Planning (Ministério do Planejamento) reported that on January 2012, over 300 000 modems were still affected [23].

This is an example how hardware providers should not only protect against CSRF, but make it impossible to remotely log in to the devices by default. The vulnerability allowed attackers to change the access passwords, and then they simply scanned, by brute force, the Brazilian IP address space for compromised modems. The vulnerability could not had been exploited if the modems had been in bridging mode instead of routing mode, but that configuration opens another can of worms. Changing the modem's IP address would had made attacking more complex, but it is possible to programmatically get host IP address [24] and then blindly enumerate the subnet.

6. CONCLUSIONS
We have, hopefully in sufficient detail, described the CSRF, its nature and associated issues and vulnerabilities. The HTTP protocol along with headers relevant to the problem, HTML, and Javascript are explained in necessary detail to understand the machinery under the hood. Known, tried, failed and working defenses against CSRF and login CSRF are described along with history and somewhat famous examples of exploits of this vulnerability. The original content is limited to analysis of past and current attempts, historical cases of some significance and their analysis, impact and reasons, the realization that existing Referer header is a usable defense and finally the attempt to explain how any web application could be used in a secure manner.

It is the author's opinion that the vulnerability will be present as long as web frameworks don't protect against it automatically. Even big players, such as Google and Facebook, have not been - and might still not be - immune to this vulnerability. Previous examples are aplenty.

CSRF has been widely exploited for about past ten years. Unless the developers build web applications with security in mind, and not only CSRF, but all other vulnerabilities too, the situation won't change.

For CSRF, it looks like an old trick is as good as a bagful of new ones. Using strict checking of Referer-header over HTTPS (or possibly even over plain HTTP) is enough to fix CSRF when implemented correctly. Apart from that, CORS support will also solve the problem when implemented correctly.

Much of the security is implemented in the browser. Because they contain bugs or differing interpretations of standards and recommendations, the web applications can't rely on browser security alone, but must be secure by design.

7. REFERENCES
[1] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). ACM SIGOPS Operating Systems


[28] A. Barth. HTTP State Management Mechanism. IETF Request For Comments, RFC 6265, April 2011.


A. Klein. Exploiting the XMLHttpRequest object in IE - Referrer spoofing, and a lot more. ... http://www.cgisecurity.com/lib/XMLHttpRequest.shtml, September 2005


