Cross Site Request Forgery

Konstantin Käfer

Hasso-Plattner-Institut, Potsdam

Abstract. *Cross Site Request Forgery* is a technique that allows the attacker to trick a user into performing an action, using her authority and credentials. Another commonly used name is “Session Riding”. It works by submitting HTTP requests from the user’s web browser to a vulnerable web application. The victim’s browser automatically sends the authorization information stored in the cookie with the request; if the victim is logged into a web application, the requested action is performed because the web application does not know that this was a forged request.

1 The Confused Deputy attack

Cross Site Request Forgery is commonly referred to as a “Confused Deputy” attack. In this type of attack, a system is exploited by tricking it into using its authority to do something on the behalf of the attacker. The targeted system (the “confused deputy”) is allowed to perform the action the client requests, but the way the system is used was not intended.

The Confused Deputy problem was first described by Norm Hardy ([1]). The original example explained a compilation service that maintained a billing file where it stored billing information for the clients. Regular users don’t have (write) access to that billing file so that they can’t manipulate the billings file. However, it turned out that users could supply arbitrary output paths for the compilation results. Since the compilation system itself could edit the billings file (since it had to record who compiled how many files to be able to bill the clients later) it could also overwrite the billings file with the compilation results, thus destroying the billings file effectively.

This is a good description of a Confused Deputy attack in which the attacker (the clients who want to compile files) can trick the compilation system into destroying its billings file. This is not necessarily a security hole; the compilation system was *supposed* to be able to write to the billings file, but not in the way the client requested it.

Cross Site Request Forgeries are similar: The user does in fact have the permission to do a certain thing on a web page, like buying a book, changing her password or transferring funds since she is logged into the web application that allows her to perform this task. The only difference is that the request does not originate from a web page generated by that system but from a third party web page, hosted on another computer.

2 The Browser Security Model

To understand why a Cross Site Request Forgery works, we are going to have a look at the browser security model. Browser Security is based on the “Same Origin” principle ([2]); it’s only possible to programmatically access data in a document if it origins from the same host name as the accessing document is from. If a web page is loaded from “example.com”, it cannot access the *contents* of a web page loaded from “example.net”. It is however possible to *load* a URL from another host, for example by including it as image or as IFrame.

The Same Domain Origin model also applies to cookies. A script in a web page can only access cookies set for the domain the web page (not the script!) was loaded from. This means that you can't steal Session IDs or other values stored in cookies. Throughout the history of browsers, some security holes existed which allowed malicious scripts to access content from other hosts or read out arbitrary cookies, but they are largely gone.

The *XMLHttpRequest* object introduced by Internet Explorer and adopted by all major browser vendors has the same restrictions: It is only possible to send requests to the same host name as the page was loaded from ([3]).

3 HTTP and Cookies

The Hypertext Transfer Protocol (HTTP 1.1 defined in RFC 2616 ([4]) is the most current version) is the predominant protocol used for retrieving web pages. It supports different request modes; GET and POST are the ones most commonly used on the web. The RFC describes it as follows: "The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI." [4, Section 9.3]. GET methods are not supposed to have any side effects on the server:

> In particular, the convention has been established that the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered 'safe'.
> (RFC 2616 [4], Section 9.1.1)

POST on the other hand is a request scheme that allows submission of data to the server in a dedicated Request Body. Both GET and POST allow the inclusion of parameters in the Request URI, but only POST requests may have side effects, such as data creation, manipulation or deletion on the server. Since a Cross Site Request Forgery usually involves manipulation of data on the server, most of forged requests are POST requests. The next section describes attack vectors and explains how POST requests can be automatically sent without user interaction from within the victim's browser.

An extension to HTTP are cookies, originally defined in RFC 2109 ([5]) and developed by Netscape. Cookies are a way to keep track of the status of a sequence of HTTP requests; The RFC is titled "HTTP State Management Mechanism". HTTP was designed as a stateless protocol, but developers needed a way to identify individual users and keep track of the state of a session. Cookies may contain arbitrary data, but their size is generally limited. Therefore, most web applications only store a unique identifier (usually called the Session ID) in the cookie and store the actual data associated with a session on the server.

Cookies can be set for host names (or with wildcard cookies for all hosts located below in a domain) and can optionally have an expiration date. Many web applications set an expiration date in the far future to save users from logging in every time they want to use the web application or website. Whenever the browser sends a request to a server for which it has stored a cookie, it sends the cookie contents along in the "`Cookie:`" HTTP header. It does not matter how the request was initiated; even if a web page from another host initiates a request (e.g. by including a specific URL as image in its page), the cookie is sent along.

4 Attack Vectors

Cross Site Request Forgeries can be performed in several different ways, depending on how strong the attacked web application or website is protected against such attacks.

4.1 Images

The simplest Cross Site Request Forgery attack requires a poorly implemented web applications that perform actions for GET requests as target. The attacker prepares a web page which contains an image with the image's source attribute set to the vulnerable URL, e.g.:

```
<img src="http://example.com/transfer_funds?destination=123456&
    amount=1000">
```

As soon as the victim loads the web page containing this code, a request to the specified URL will be performed and the action is executed on the server.

An easy and very effective way to protect users against this simple attack is to only allow data change operations for POST requests. If it is for some reason not possible to convert these operations to POST requests, a token can be included in the URL to verify that the request was in fact initiated from a valid source. The section "Prevention Mechanisms" outlines the requirements for tokens in greater detail.

4.2 Autosubmitting Forms

The only way to perform POST requests to another host is to use the HTML ¡form¿ tag with the action attribute set to the target URL. The required form fields should be set the desired values. Once the form is prepared, the attacker can either rely on the user to click the submit button or he can submit the form programatically by calling the DOM node's "submit" function. A form submission replaces the currently displayed website with the form action's URL. However, if the attack should remain undetected for as long as possible, it is also possible to create an IFrame and redirect the form submission into that IFrame. The IFrame can be hidden with CSS so that the user does not notice that a form submission has taken place in the background.

Listing 1.1. Autosubmitting form

```
<iframe name="frm" style="display:none"></iframe>

<form action="http://example.com/transfer_funds" method="POST"
    target="frm">
    <input type="text" name="destination" value="123456">
    <input type="text" name="amount" value="1000">
</form>

<script type="text/javascript">
    document.forms[0].submit();
</script>
```

When the form is submitted, the POST request to the specified URL is performed and the browser sends the cookie for that host name along with the request. The attack succeeds when the victim visits the page the attacker prepared and is logged into the abused system. A way to protect form submissions originating from non-authorized forms is to include a unique token in generated forms and only perform data change operations when the submitted token matches the token that was created when generating the page.

4.3 XMLHttpRequest

From an attacker's point of view, the XMLHttpRequest object available in many browser (and commonly used to perform AJAX communication) is very interesting because it allows setting arbitrary HTTP headers and has the added benefit of being able to read the result as opposed to just sending the request. XMLHttpRequest can be used in environments where an attacker has access to one subdomain, user.example.com and the management of the service happens on the parent domain, example.com. By setting "`document.domain`" to "`example.com`" in the JavaScript code running in the user.example.com context, requests to example.com can be performed ([6]).

Cross Site Request Forgery attacks require the victim to go to a certain URL, but they are still exploitable in a large scale. As an example, a combination of XSS (Cross Site Scripting) and CSRF was used in 2005 to create a MySpace worm named "Samy" Within a day, a user managed to get over 1 million friends on MySpace by inserting a code snippet on his profile page that made the viewer add him as a friend in the background. Not only did it do that, but it also added the same code to the profile page of the viewer so the malicious code spread around the large parts of the website within only a couple of hours until MySpace took down the site to repair the damage done and fix the security hole. Technically, this attack type is not "Cross Site" because it didn't cross host name boundaries, but the principle is very similar ([7]).

4.4 Flash and Java

Browser Plugins like Flash and Java often times allow developers to perform requests as well. However, they generally also implement the Same Origin Security model. Flash also supports a way to perform Cross Domain Requests, specified by Adobe ([8]). The Flash Player requests the "`crossdomain.xml`" file at the root of a hostname when a Flash movie tries to access content on this domain. Based on the policies in this file, Flash Player allows or restricts the access. If the file is not present, access is prohibited in general. While it's a good idea to allow developers to choose which host names have access to their host name, it can also open potential security holes. Even YouTube had a wildcard `crossdomain.xml` on their servers until late 2006 until they have been notified that it's technically possible that malicious Flash movies perform arbitrary actions on their servers with the user's login data. Typical CSRF protections (like tokens) did not prevent this since it was possible to retrieve tokens because Flash allowed the Movie to access the content retrieved.

To make a forged request, it is necessary that the user retains his authentication data acquired via cookies from the browser. The Flash player does just this: It retrieves the cookies from the browser and sends them with every request. Java(tm) also applies the Same Origin restrictions to requests ([11]), but it is possible to acquire the authorization from the user to contact arbitrary hosts (and in fact use arbitrary networking technologies, not just HTTP).

5 Variants

Cross Site Request Forgeries allow for a large variety of attacks. The most commonly implemented attack is probably a straight session riding attack that exploits the fact that the user is already logged into a web site. This includes all kinds of malicious activities if the web application is not properly protected. Since CSRF is not as well known as other attacks against web applications, there is a considerable amount of web applications out there which do not have any protection whatsoever. Most major websites are protected, but even those occasionally have CSRF holes. In November 2008, a domain fraud victim claimed that a CSRF hole in Google's GMail web application allowed an attacker to create a filter that actually didn't filter anything but forwarded the filtered results (hence all mails) to the attacker. That hacker could then send a "Password forgotton" request and obtain the login data to transfer a domain the victim owned to himself.

A more subtle attack tries to modify the configuration settings of the user's router. Often times, the configuration menu is freely accessible from within the local network or only protected with a weak password. An attacker could use the fact that the requests are performed from the user's browser, which is located inside the network, to change for example the DNS server in the router configuration menu to one the attacker controls. Subsequently, he could manipulate the DNS entries so that various web pages point to his own copy. Using this approach, it is also possible to perform IP and port scanning inside a network to reveal more about the internal structure to the external attacker.

A third, slightly different attack is referred to as "logout/login attack". In this scenario, the user is logged out of a web application and logged in with the attackers credentials. The user does not notice this since the logout/login happens in the background. If the user then enters personal data into the allegedly trusted website, the attacker is able to retrieve this information later on.

A fourth scenario involves a local HTTP server installed on the victim's computer. This server could be from a regular application program such as Google Desktop Search or μTorrent (which provides a web interface for managing torrents). These web servers usually have write permissions on the victim's computer. In fact, such an hole existed in μTorrent ([9]). The attacker abuses μTorrent's web server to download a faked torrent file and store it in the user's Autostart folder so that the faked torrent (which is actually an executable file) is started when the user reboots the system. This attack circumvents the firewall since the attack is local to the system and does not cross machine boundaries.

6 Prevention Mechanisms

There are various ways to prevent Cross Site Request Forgery attacks. The most naïve is to only allow data to be changed on POST requests. However, since arbitrary POST requests can be performed using the Autosubmission technique explained in the "Attack Vectors" section, this is not an effective protection, but can only prevent the simplest attacks.

6.1 Checking Referrer headers

Most browser send a "`Referer:`" HTTP header with each request which contains the URL the request originates from (e.g. the page the link the user clicked on is from). This could be used to verify that the request does indeed originate from a web page on the same host name. In general, this suffices to provide an adequate protection, but there are several caveats:

- Not all browsers send Referrer headers and this can be turned off by the user
- Referrer headers could be forged; there have been security holes in the past in the Flash player which made this possible
- Referrer headers are not send for HTTPS requests due to privacy implications

This means that while a Referrer header check is better than no protection, it does not work in all situations and should therefore be avoided when a better solution is readily available.

6.2 Session ID in URL

Cross Site Request Forgeries are usually based on the browser sending the cookie belonging to the site along with the request. When there is no cookie, the user is not authenticated and therefore cannot perform malicious actions. However, dynamic web applications require a way to identify and track individual users. This ID is stored in the cookie, but it could also be stored in the URLs instead. This means that each URL contains the Session ID. An attacker is unable to guess the Session ID and can therefore not perform forged requests. Unfortunately, there are several caveats as well:

– A user could publish the URL containing his session ID and an attacker could use this session ID to gain access to the user's account
– The URL becomes confusing to end users

While this is an effective protection against forged requests, the risk for accidentally revealed Session IDs makes this technique impracticable.

6.3 Tokens/Nonces

Another way to prevent arbitrary requests to a URL is to require a nonce to be present. This nonce or token is generated along with the form on the permitted website. All requests that lack the token or provide an invalid token are rejected. This ensures that requests are only accepted from authorized origin web pages since the attacker is unable to guess the token.

Randomly created tokens have to be stored on the server to verify that this token indeed exists. There should be a time limit for each token so that leaked tokens cannot be used after a certain amount of time. The disadvantage of this mechanism is the large amount of stale and unused tokens stored on the server.

A better approach is to create tokens algorithmically. A token generated in this way should fulfill these requirements:

– Keyed to a specific action. This ensures that a token can only be used for that particular action. If an attacker learns about a token, he can only use it for this action and is unable to cause further damage. This also means that a particular action should only be executable once.
– Keyed to a specific user (or his session ID). This makes sure that a token for the same action is different for each user. The attacker is therefore unable to reuse the token acquired from his own account.
– Keyed to a specific web application. A randomly created private key for the web site should be incorporated when creating tokens. This ensures that the attacker cannot regenerate tokens for a specific action ID and user ID. This is not necessary when the unique session ID is used instead of a guessable user ID.
– Ideally, tokens should also be dependent on the time so that tokens become invalid even though neither the session ID nor the private key or the action ID change.

This is the most secure way to protect a web application from Cross Site Request Forgery attacks, but it it also the most complex one.

6.4 Cookie as part of POST body

An easier way to ensure that requests originate from the same host is to read out the cookie (which is restricted by the Same Origin policies) and insert it as field in the form before submission. The web application then has to check whether the submitted session ID in the POST body matches the session ID sent in the HTTP header. The attacker is unable to read the cookie since the web page is generally loaded from a different host name.

This technique has the disadvantage of requiring JavaScript to function, but it can be used when generating tokens is impractical and JavaScript is available anyway, for example in Rich Client applications which employ `XMLHttpRequest` to perform actions of the server.

6.5 Origin HTTP header

The `Origin` HTTP header is a mechanism proposed by Barth, Jackson and Mitchell in their paper "Robust Defenses for Cross-Site Request Forgery" ([10]). Browsers should be modified to send an "`Origin`" HTTP header when performing a POST request. The origin of a request is the host name of the page which performs the request was loaded from. In contrast to the referrer header, the origin header should not contain the path name part of the URL, thus mitigating privacy concerns. The disadvantage of this technique is that currently no browser implements the Origin header and until most clients support it, it cannot be used as a mean to prevent Cross Site Request Forgeries.

7 Conclusion

Cross Site Request Forgery holes revealed in the past have shown that this attack is not theoretical in any way but can be exploited in real life without too much effort. A large amount of web applications are susceptible to this kind of attack because it is not known widely, unlike Cross Site Scripting or SQL injection. Browser vendors could also play a major role in limiting the extent of Cross Site Request Forgeries by not sending cookies for cross domain requests unless the user requests it.

References

1. Hardy, N.: The confused deputy: (or why capabilities might have been invented). SIGOPS Oper. Syst. Rev. **22**(4) (1988) 36–38
2. Zalewski, M.: Browser security handbook; same-origin policy [`http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy` accessed on Dec 15, 2008].
3. Mozilla Foundation: Same origin policy for JavaScript. [`https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`; accessed on Dec 01, 2008].
4. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. Technical Report 2616 (June 1999) Updated by RFC 2817.
5. Kristol, D., Montulli, L.: HTTP State Management Mechanism. Technical Report 2109 (February 1997) Obsoleted by RFC 2965.
6. Mozilla Foundation: document.domain. [`https://developer.mozilla.org/en/DOM/document.domain`; accessed on Dec 01, 2008].
7. samy: Technical explanation of the myspace worm [`http://namb.la/popular/tech.html` accessed on Dec 01, 2008].
8. Adobe, Inc.: Cross-domain policy file specification. [`http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html`; accessed on Dec 01, 2008].
9. Rios, B.: {CSRF pwns your box [`http://xs-sniper.com/blog/2008/04/21/csrf-pwns-your-box/` accessed on Dec 01, 2008].
10. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. (2008) [`http://crypto.stanford.edu/websec/csrf/csrf.pdf`; accessed on Dec 01, 2008].
11. Sun, Inc.: Security considerations for Java Applets. [`http://java.sun.com/j2se/1.3/docs/guide/plugin/security.html`; accessed on Dec 09, 2008].