**Report of CSRF and CORS**

**What is CSRF?**

Cross-site request forgery (also known as CSRF) is a web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform. It allows an attacker to partly circumvent the same origin policy, which is designed to prevent different websites from interfering with each other.

**Problem Statement**

* A relevant action. There is an action within the application that the attacker has a reason to induce. This might be a privileged action (such as modifying permissions for other users) or any action on user-specific data (such as changing the user's own password).
* Cookie-based session handling. Performing the action involves issuing one or more HTTP requests, and the application relies solely on session cookies to identify the user who has made the requests. There is no other mechanism in place for tracking sessions or validating user requests.
* No unpredictable request parameters. The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess. For example, when causing a user to change their password, the function is not vulnerable if an attacker needs to know the value of the existing password.

**Available Solutions**

1. **Use Built-In Or Existing CSRF Implementations for CSRF Protection.**

Synchronizer token defenses have been built into many frameworks. It is strongly recommended to research if the framework you are using has an option to achieve CSRF protection by default before trying to build your custom token generating system. For example, .NET has build-in protection that adds a token to CSRF vulnerable resources. You are responsible for proper configuration (such as key management and token management) before using these built-in CSRF protections that generate tokens to guard CSRF vulnerable resources.

**2) Synchronizer Token Pattern**

CSRF tokens should be generated on the server-side. They can be generated once per user session or for each request. Per-request tokens are more secure than per-session tokens as the time range for an attacker to exploit the stolen tokens is minimal. However, this may result in usability concerns. For example, the "Back" button browser capability is often hindered as the previous page may contain a token that is no longer valid. Interaction with this previous page will result in a CSRF false positive security event at the server. In per-session token implementation after initial generation of token, the value is stored in the session and is used for each subsequent request until the session expires.

CSRF tokens should be:

* Unique per user session.
* Secret
* Unpredictable (large random value generated by a secure method.

**3) CSRF tokens should not be transmitted using cookies**.

The CSRF token can be added through hidden fields, headers, and can be used with forms, and AJAX calls. Make sure that the token is not leaked in the server logs, or in the URL. CSRF tokens in GET requests are potentially leaked at several locations, such as the browser history, log files, network appliances that log the first line of an HTTP request, and Referer headers if the protected site links to an external site.

**4) Double Submit Cookie**

we send a random value in both a cookie and as a request parameter, with the server verifying if the cookie value and request value match. When a user visits (even before authenticating to prevent login CSRF), the site should generate a (cryptographically strong) pseudorandom value and set it as a cookie on the user's machine separate from the session identifier. The site then requires that every transaction request include this pseudorandom value as a hidden form value (or other request parameter/header). If both of them match at server side, the server accepts it as legitimate request and if they don't, it would reject the request.

**5) SameSite Cookie Attribute**

SameSite is a cookie attribute. which aims to mitigate CSRF attacks. It is defined in RFC6265bis. This attribute helps the browser decide whether to send cookies along with cross-site requests. Possible values for this attribute are Lax, Strict, or None.

**Strict**: As the name suggests, this is the option in which the Same-Site rule is applied strictly. When the SameSite attribute is set as Strict, the cookie will not be sent along with requests initiated by third party websites.

**Lax**: When you set a cookie' SameSite attribute to Lax, the cookie will be sent along with the GET request initiated by third party website.

Set-Cookie: CookieName=CookieValue; SameSite=Lax;

Set-Cookie: CookieName=CookieValue; SameSite=Strict

**7) User Interaction Based CSRF Defense**

While all the techniques referenced here do not require any user interaction, sometimes it's easier or more appropriate to involve the user in the transaction to prevent unauthorized operations (forged via CSRF or otherwise). The following are some examples of techniques that can act as strong CSRF defense when implemented correctly.

* password or stronger
* One-time Token
* CAPTCHA (prefer newer CAPTCHA versions without user interaction or visual pattern matching)

**Advantages:**

CSRF tokens can prevent CSRF attacks by making it impossible for an attacker to construct a fully valid HTTP request suitable for feeding to a victim user.

**Disadvantages:**

The attacker must target either a site that doesn't check the referer header or a victim with a browser or plugin that allows referer spoofing.

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).

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).

The attacker must lure the victim to a web page with malicious code while the victim is logged into the target site.

**Best solution:**

A key design principle that protects you from CSRF attacks is using GET requests for only view or read-only actions. These types of requests should not transform data and must only display recorded data. This limits the number of requests that are vulnerable to CSRF attacks.

### Anti-forgery tokens

Your website will also need to use POST, PUT, PATCH and DELETE requests. To safeguard these endpoints, you can introduce an anti-forgery token in every request that uniquely identifies safe origin sites.

Every response rendered by the server will contain the anti-forgery token which is then written out to a hidden HTML field. This token is used by the client side to authenticate requests sent to the server. Now, the server knows the request is from a safe origin.

Most modern web frameworks include anti-forgery token management, out-of-the-box.

**Example:**

1) Attacker observes URL request format:

The attacker observes that purchase requests on the website are in this format.The request assumes that the user has an open session with the website. It uses an address ID to reference an address defined by the legitimate user.

GET

https://examplebuy.com/shop/purchase?productid=3441&amount=200&address=33&20Park%20Drive%20NY%20NY HTTP/1.1

2) Attacker crafts a forged request URL:

GET

https://examplebuy.com/shop/purchase?productid=**5776**&amount=**2000**&address=**45%20Main%20Street%20NJ%20NY** HTTP/1.1

The attacker manipulates three parameters in the request—changing the product to a product they want to buy, changing the amount, and using their own address.

3) Attacker hides the URL in an image:

<img src = “https://examplebuy.com/shop/purchase?productid=5776&amount=2000&address=45%20Main%20Street%20NJ%20NY” width=“0” height= “0”>

4) Attacker uses social engineering to get the user to load the image:

The attacker sends a phishing email to the victim, which either directly includes the image, or includes a link to a web page that embeds the malicious image tag. The URL is loaded on the user’s device.

5) Ecommerce site receives the forged request:

Assuming that the user has an active session with the ecommerce site, when the URL is loaded, the website receives the forged purchase request. The website cannot identify that the request was not made directly by the legitimate user. It obeys the request and sends the goods to the attacker, billing the legitimate user’s account.

**CORS:**

**Cross-Origin Resource Sharing** (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources. CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

### **[CORS headers](https://developer.mozilla.org/en-US/docs/Glossary/CORS" \l "cors_headers)**

Access-Control-Allow-Origin : Indicates whether the response can be shared.

Access-Control-Allow-Credentials : Indicates whether or not the response to the request can be exposed when the credentials flag is true.

Access-Control-Allow-Headers : Used in response to a preflight request to indicate which HTTP headers can be used when making the actual request.

Access-Control-Allow-Methods : Specifies the method or methods allowed when accessing the resource in response to a preflight request.

Access-Control-Expose-Headers : Indicates which headers can be exposed as part of the response by listing their names.

Access-Control-Max-Age : Indicates how long the results of a preflight request can be cached.

Access-Control-Request-Headers : Used when issuing a preflight request to let the server know which HTTP headers will be used when the actual request is made.

Access-Control-Request-Method : Used when issuing a preflight request to let the server know which HTTP method will be used when the actual request is made.

Origin : Indicates where a fetch originates from.

### **Simple requests (GET, POST, and HEAD)**

The browser deems the request to be a ["simple" request](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS" \l "Simple_requests) when the request itself meets a certain set of requirements:

* One of these methods is used: GET, POST, or HEAD
* A [CORS safe-listed header](https://fetch.spec.whatwg.org/" \l "cors-safelisted-request-header) is used
* When using the Content-Type header, only the following values are allowed: application/x-www-form-urlencoded, multipart/form-data, or text/plain
* No event listeners are registered on any XMLHttpRequestUpload object
* No ReadableStream object is used in the request

The request is allowed to continue as normal if it meets these criteria, and the Access-Control-Allow-Origin header is checked when the response is returned.

### **Preflight requests (OPTIONS)**

If a request does not meet the criteria for a simple request, the browser will instead make an automatic [preflight request](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS" \l "Preflighted_requests) using the OPTIONS method. This call is used to determine the exact CORS capabilities of the server, which is in turn used to determine whether or not the intended CORS protocol is understood. If the result of the OPTIONS call dictates that the request cannot be made, the actual request to the server will not be executed.

The preflight request sets the mode as OPTIONS and sets a couple of headers to describe the actual request that is to follow:

* Access-Control-Request-Method: The intended method of the request (e.g., GET or POST)
* Access-Control-Request-Headers: An indication of the custom headers that will be sent with the request
* Origin: The usual origin header that contains the script's current origin

**Example:**

browser will send to the server :

GET /resources/public-data/ HTTP/1.1

Host: bar.other

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0) Gecko/20100101 Firefox/71.0

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Connection: keep-alive

Origin: https://foo.example

The request header of note is Origin, which shows that the invocation is coming from https://foo.example.

HTTP/1.1 200 OK

Date: Mon, 01 Dec 2008 00:23:53 GMT

Server: Apache/2

Access-Control-Allow-Origin: \*

Keep-Alive: timeout=2, max=100

Connection: Keep-Alive

Transfer-Encoding: chunked

Content-Type: application/xml

References:

<https://www.imperva.com/learn/application-security/csrf-cross-site-request-forgery/>

<https://auth0.com/blog/cors-tutorial-a-guide-to-cross-origin-resource-sharing/>

<https://en.wikipedia.org/wiki/Cross-origin_resource_sharing>

<https://owasp.org/www-community/attacks/csrf>