Cross-Site Request Forgery Prevention Cheat Sheet

Introduction

Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious web site, email, blog, instant message, or program causes a user's web browser to perform an unwanted action on a trusted site when the user is authenticated. A CSRF attack works because browser requests automatically include all cookies including session cookies. Therefore, if the user is authenticated to the site, the site cannot distinguish between legitimate authorized requests and forged authenticated requests. This attack is thwarted when proper Authorization is used, which implies that a challenge-response mechanism is required that verifies the identity and authority of the requester.

The impact of a successful CSRF attack is limited to the capabilities exposed by the vulnerable application and privileges of the user. For example, this attack could result in a transfer of funds, changing a password, or making a purchase with the user's credentials. In effect, CSRF attacks are used by an attacker to make a target system perform a function via the victim's browser, without the victim's knowledge, at least until the unauthorized transaction has been committed.

In short, the following principles should be followed to defend against CSRF:

- **Check if your framework has built-in CSRF protection and use it**
  - **If framework does not have built-in CSRF protection add CSRF tokens to all state changing requests (requests that cause actions on the site) and validate them on backend**
- **For stateful software use the synchronizer token pattern**
- **For stateless software use double submit cookies**
- **Implement at least one mitigation from Defense in Depth Mitigations section**
  - **Consider SameSite Cookie Attribute for session cookies** but be careful to NOT set a cookie specifically for a domain as that would introduce a security vulnerability that all subdomains of that domain share the cookie. This is particularly an issue when a subdomain has a CNAME to domains not in your control.
  - **Consider implementing user interaction based protection for highly sensitive operations**
  - **Consider the use of custom request headers**
  - **Consider verifying the origin with standard headers**
- **Remember that any Cross-Site Scripting (XSS) can be used to defeat all CSRF mitigation techniques!**
  - **See the OWASP XSS Prevention Cheat Sheet for detailed guidance on how to prevent XSS flaws.**
- **Do not use GET requests for state changing operations.**
  - **If for any reason you do it, protect those resources against CSRF**

Token Based Mitigation

The synchronizer token pattern is one of the most popular and recommended methods to mitigate CSRF.

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 built-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.

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.

When a request is issued by the client, the server-side component must verify the existence and validity of the token in the request compared to the token found in the user session. If the token was not found within the

request, or the value provided does not match the value within the user session, then the request should be aborted, session of the user terminated and the event logged as a potential CSRF attack in progress.

CSRF tokens should be:

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

CSRF tokens prevent CSRF because without token, attacker cannot create a valid requests to the backend server.

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.

For example:

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken"
value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWEwYzU1YWQwMTVhM2JmNGYxYjJ

[...]
</form>
```

Inserting the CSRF token in the custom HTTP request header via JavaScript is considered more secure than adding the token in the hidden

field form parameter because it uses custom request headers.

Double Submit Cookie

If maintaining the state for CSRF token at server side is problematic, an alternative defense is to use the double submit cookie technique. This technique is easy to implement and is stateless. In this technique, 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.

Because subdomains can write cookies to the parent domains and because cookies can be set for the domain over plain HTTP connections this technique works as long as you are sure that your subdomains are fully secured and only accept HTTPS connections.

To enhance the security of this solution include the token in an encrypted cookie - other than the authentication cookie (since they are often shared within subdomains) - and then at the server side match it (after decrypting the encrypted cookie) with the token in hidden form field or parameter/header for AJAX calls. This works because a sub domain has no way to over-write an properly crafted encrypted cookie without the necessary information such as encryption key.

A simpler alternative to an encrypted cookie is to HMAC the token with a secret key known only by the server and place this value in a cookie. This is similar to an encrypted cookie (both require knowledge only the server holds), but is less computationally intensive than encrypting and decrypting the cookie. Whether encryption or a HMAC is used, an attacker won't be able to recreate the cookie value from the plain token without knowledge of the server secrets.

Defense In Depth Techniques

SameSite Cookie Attribute

SameSite is a cookie attribute (similar to HTTPOnly, Secure etc.) 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`.

The Strict value will prevent the cookie from being sent by the browser to the target site in all cross-site browsing context, even when following a regular link. For example, for a GitHub-like website this would mean that if a logged-in user follows a link to a private GitHub project posted on a corporate discussion forum or email, GitHub will not receive the session cookie and the user will not be able to access the project. A bank website however doesn't want to allow any transactional pages to be linked from external sites, so the Strict flag would be most appropriate.

The default Lax value provides a reasonable balance between security and usability for websites that want to maintain user's logged-in session after the user arrives from an external link. In the above GitHub scenario, the session cookie would be allowed when following a regular link from an external website while blocking it in CSRF-prone request methods

such as POST. Only cross-site-requests that are allowed in Lax mode are the ones that have top-level navigations and are also safe HTTP methods.

For more details on the `SameSite` values, check the following section from the rfc.

Example of cookies using this attribute:

```
Set-Cookie: JSESSIONID=xxxxx; SameSite=Strict
Set-Cookie: JSESSIONID=xxxxx; SameSite=Lax
```

All desktop browsers and almost all mobile browsers now support the `SameSite` attribute. To keep track of the browsers implementing it and the usage of the attribute, refer to the following service. Note that Chrome has announced that they will mark cookies as `SameSite=Lax` by default from Chrome 80 (due in February 2020), and Firefox and Edge are both planning to follow suit. Additionally, the `Secure` flag will be required for cookies that are marked as `SameSite=None`.

It is important to note that this attribute should be implemented as an additional layer *defense in depth* concept. This attribute protects the user through the browsers supporting it, and it contains as well 2 ways to bypass it as mentioned in the following section. This attribute should not replace having a CSRF Token. Instead, it should co-exist with that token in order to protect the user in a more robust way.

Verifying Origin With Standard Headers

There are two steps to this mitigation, both of which rely on examining an HTTP request header value.

1. Determining the origin the request is coming from (source origin). Can be done via Origin or Referer headers.
2. Determining the origin the request is going to (target origin).

At server side we verify if both of them match. If they do, we accept the request as legitimate (meaning it's the same origin request) and if they don't, we discard the request (meaning that the request originated from cross-domain). Reliability on these headers comes from the fact that they cannot be altered programmatically (using JavaScript with an XSS vulnerability) as they fall under forbidden headers list, meaning that only the browser can set them

Identifying Source Origin (via Origin/Referer header)

CHECKING THE ORIGIN HEADER

If the Origin header is present, verify that its value matches the target origin. Unlike the Referer, the Origin header will be present in HTTP requests that originate from an HTTPS URL.

CHECKING THE REFERER HEADER

If the Origin header is not present, verify the hostname in the Referer header matches the target origin. This method of CSRF mitigation is also commonly used with unauthenticated requests, such as requests made prior to establishing a session state, which is required to keep track of a synchronization token.

In both cases, make sure the target origin check is strong. For example, if your site is `example.org` make sure `example.org.attacker.com` does not pass your origin check (i.e, match through the trailing / after the origin to make sure you are matching against the entire origin).

If neither of these headers are present, you can either accept or block the request. We recommend **blocking**. Alternatively, you might want to log all such instances, monitor their use cases/behavior, and then start blocking requests only after you get enough confidence.

Identifying the Target Origin

You might think it's easy to determine the target origin, but it's frequently not. The first thought is to simply grab the target origin (i.e., its hostname and port `#`) from the URL in the request. However, the application server is frequently sitting behind one or more proxies and the original URL is different from the URL the app server actually receives. If your application server is directly accessed by its users, then using the origin in the URL is fine and you're all set.

If you are behind a proxy, there are a number of options to consider.

- **Configure your application to simply know its target origin:** It's your application, so you can find its target origin and set that value in some server configuration entry. This would be the most secure approach as it's defined server side, so it is a trusted value. However, this might be problematic to maintain if your application is deployed in many places, e.g., dev, test, QA, production, and possibly multiple production instances. Setting the correct value for each of these situations might be difficult, but if you can do it via some central configuration and providing your instances to grab value from it, that's great! (**Note:** Make sure the centralized configuration store is maintained securely because major part of your CSRF defense depends on it.)
- **Use the Host header value:** If you prefer that the application find its own target so it doesn't have to be configured for each deployed instance, we recommend using the Host family of headers. The Host

header's purpose is to contain the target origin of the request. But, if your app server is sitting behind a proxy, the Host header value is most likely changed by the proxy to the target origin of the URL behind the proxy, which is different than the original URL. This modified Host header origin won't match the source origin in the original Origin or Referer headers.

- **Use the X-Forwarded-Host header value:** To avoid the issue of proxy altering the host header, there is another header called X-Forwarded-Host, whose purpose is to contain the original Host header value the proxy received. Most proxies will pass along the original Host header value in the X-Forwarded-Host header. So that header value is likely to be the target origin value you need to compare to the source origin in the Origin or Referer header.

This mitigation is working properly when origin or referrer headers are present in the requests. Though these headers are included **majority** of the time, there are few use cases where they are not included (most of them are for legitimate reasons to safeguard users privacy/to tune to browsers ecosystem). The following lists some use cases:

- Internet Explorer 11 does not add the Origin header on a CORS request across sites of a trusted zone. The Referer header will remain the only indication of the UI origin. See the following references in Stack Overflow here and here.
- In an instance following a 302 redirect cross-origin, Origin is not included in the redirected request because that may be considered sensitive information that should not be sent to the other origin.
- There are some privacy contexts where Origin is set to "null" For example, see the following here.

- Origin header is included for all cross origin requests but for same origin requests, in most browsers it is only included in POST/DELETE/PUT **Note:** Although it is not ideal, many developers use GET requests to do state changing operations.
- Referer header is no exception. There are multiple use cases where referrer header is omitted as well (1, 2, 3, 4 and 5). Load balancers, proxies and embedded network devices are also well known to strip the referrer header due to privacy reasons in logging them.

Usually, a minor percentage of traffic does fall under above categories (1-2%) and no enterprise would want to lose this traffic. One of the popular technique used across the Internet to make this technique more usable is to accept the request if the Origin/referrer matches your configured list of domains "OR" a null value (Examples here. The null value is to cover the edge cases mentioned above where these headers are not sent). Please note that, attackers can exploit this but people prefer to use this technique as a defense in depth measure because of the minor effort involved in deploying it.

Cookie with __Host- prefix

Another solution for this problem is use of `Cookie Prefixes` for cookie with CSRF token. If cookie has `__Host-` prefix e.g. `Set-Cookie: __Host-token=RANDOM; path=/; Secure` then the cookie:

- Cannot be (over)written from another subdomain.
- Must have the path of `/`.
- Must be marked as Secure (i.e, cannot be sent over unencrypted HTTP).

As of July 2020 cookie prefixes are supported by all major browsers except Internet Explorer.

See the Mozilla Developer Network and IETF Draft for further information about cookie prefixes.

Use of Custom Request Headers

Adding CSRF tokens, a double submit cookie and value, an encrypted token, or other defense that involves changing the UI can frequently be complex or otherwise problematic. An alternate defense that is particularly well suited for AJAX or API endpoints is the use of a **custom request header**. This defense relies on the same-origin policy (SOP) restriction that only JavaScript can be used to add a custom header, and only within its origin. By default, browsers do not allow JavaScript to make cross origin requests with custom headers.

If this is the case for your system, you can simply verify the presence of this header and value on all your server side AJAX endpoints in order to protect against CSRF attacks. This approach has the double advantage of usually requiring no UI changes and not introducing any server side state, which is particularly attractive to REST services. You can always add your own **custom header** and value if that is preferred.

This technique obviously works for AJAX calls, but you still need to protect `<form>` tags with approaches described in this document such as tokens. Also, CORS configuration should also be robust to make this solution work effectively (as custom headers for requests coming from other domains trigger a pre-flight CORS check).

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.

- ~~Re-Authentication~~ Authorization mechanism (password or stronger)
- One-time Token
- CAPTCHA (prefer newer CAPTCHA versions without user interaction or visual pattern matching)

While these are a very strong CSRF defense, it can create a significant impact on the user experience. As such, they would generally only be used for security critical operations (such as password change, money transfers, etc.), alongside the other defences discussed in this cheat sheet.

Login CSRF

Most developers tend to ignore CSRF vulnerability on login forms as they assume that CSRF would not be applicable on login forms because user is not authenticated at that stage, however this assumption is not always true. CSRF vulnerabilities can still occur on login forms where the user is not authenticated, but the impact and risk is different.

For example, if an attacker uses CSRF to assume an authenticated identity of a target victim on a shopping website using the attacker's account, and the victim then enters their credit card information, an attacker may be able to purchase items using the victim's stored card details. For more information about login CSRF and other risks, see section 3 of this paper.

Login CSRF can be mitigated by creating pre-sessions (sessions before a user is authenticated) and including tokens in login form. You can use any of the techniques mentioned above to generate tokens. Remember that pre-sessions cannot be transitioned to real sessions once the user is authenticated - the session should be destroyed and a new one should be made to avoid session fixation attacks. This technique is described in Robust Defenses for Cross-Site Request Forgery section 4.1.

If sub-domains under your master domain are not trusted in your threat model, it is difficult to mitigate login CSRF. A strict subdomain and path level referrer header validation can be used in these cases for mitigating CSRF on login forms to an extent.

Java Reference Example

The following JEE web filter provides an example reference for some of the concepts described in this cheatsheet. It implements the following stateless mitigations (OWASP CSRFGuard, cover a stateful approach).

- Verifying same origin with standard headers
- Double submit cookie
- SameSite cookie attribute

Please note that it only acts a reference sample and is not complete (for example: it doesn't have a block to direct the control flow when origin and referrer header check succeeds nor it has a port/host/protocol level validation for referrer header). Developers are recommended to build their complete mitigation on top of this reference sample. Developers should also implement authentication and authorization mechanisms before checking for CSRF is considered effective.

Full source is located here and provides a runnable POC.

JavaScript Guidance for Auto-inclusion of CSRF tokens as an AJAX Request header

The following guidance considers **GET**, **HEAD** and **OPTIONS** methods are safe operations. Therefore **GET**, **HEAD**, and **OPTIONS** method AJAX calls need not be appended with a CSRF token header. However, if the verbs are used to perform state changing operations, they will also require a CSRF token header (although this is bad practice, and should be avoided).

The **POST**, **PUT**, **PATCH**, and **DELETE** methods, being state changing verbs, should have a CSRF token attached to the request. The following guidance will demonstrate how to create overrides in JavaScript libraries to have CSRF tokens included automatically with every AJAX request for the state changing methods mentioned above.

Storing the CSRF Token Value in the DOM

A CSRF token can be included in the `<meta>` tag as shown below. All subsequent calls in the page can extract the CSRF token from this `<meta>` tag. It can also be stored in a JavaScript variable or anywhere on the DOM. However, it is not recommended to store it in cookies or browser local storage.

The following code snippet can be used to include a CSRF token as a `<meta>` tag:

```
<meta name="csrf-token" content="{{ csrf_token() }}">
```

The exact syntax of populating the content attribute would depend on your web application's backend programming language.

Overriding Defaults to Set Custom Header

Several JavaScript libraries allow for overriding default settings to have a header added automatically to all AJAX requests.

XMLHttpRequest (Native JavaScript)

XMLHttpRequest's open() method can be overridden to set the `anti-csrf-token` header whenever the `open()` method is invoked next. The function `csrfSafeMethod()` defined below will filter out the safe HTTP methods and only add the header to unsafe HTTP methods.

This can be done as demonstrated in the following code snippet:

```
<script type="text/javascript">
    var csrf_token =
document.querySelector("meta[name='csrf-
token']").getAttribute("content");
    function csrfSafeMethod(method) {
        // these HTTP methods do not require CSRF
protection
        return (/^(GET|HEAD|OPTIONS)$/.test(method));
    }
    var o = XMLHttpRequest.prototype.open;
    XMLHttpRequest.prototype.open = function(){
        var res = o.apply(this, arguments);
        var err = new Error();
        if (!csrfSafeMethod(arguments[0])) {
            this.setRequestHeader('anti-csrf-token',
csrf_token);
        }
        return res;
```

```
    };
 </script>
```

AngularJS

AngularJS allows for setting default headers for HTTP operations. Further documentation can be found at AngularJS's documentation for $httpProvider.

```
<script>
    var csrf_token =
document.querySelector("meta[name='csrf-
token']").getAttribute("content");

    var app = angular.module("app", []);

    app.config(['$httpProvider', function ($httpProvider) {
        $httpProvider.defaults.headers.post["anti-csrf-
token"] = csrf_token;
        $httpProvider.defaults.headers.put["anti-csrf-
token"] = csrf_token;
        $httpProvider.defaults.headers.patch["anti-csrf-
token"] = csrf_token;
        // AngularJS does not create an object for DELETE
and TRACE methods by default, and has to be manually
created.
        $httpProvider.defaults.headers.delete = {
            "Content-Type" : "application/json;charset=utf-
8",
            "anti-csrf-token" : csrf_token
        };
        $httpProvider.defaults.headers.trace = {
            "Content-Type" : "application/json;charset=utf-
8",
            "anti-csrf-token" : csrf_token
        };
      }]);
 </script>
```

This code snippet has been tested with AngularJS version 1.7.7.

Axios

Axios allows us to set default headers for the POST, PUT, DELETE and PATCH actions.

```
<script type="text/javascript">
    var csrf_token =
document.querySelector("meta[name='csrf-
token']").getAttribute("content");

    axios.defaults.headers.post['anti-csrf-token'] =
csrf_token;
    axios.defaults.headers.put['anti-csrf-token'] =
csrf_token;
    axios.defaults.headers.delete['anti-csrf-token'] =
csrf_token;
    axios.defaults.headers.patch['anti-csrf-token'] =
csrf_token;

    // Axios does not create an object for TRACE method by
default, and has to be created manually.
    axios.defaults.headers.trace = {}
    axios.defaults.headers.trace['anti-csrf-token'] =
csrf_token
</script>
```

This code snippet has been tested with Axios version 0.18.0.

JQuery

JQuery exposes an API called $.ajaxSetup() which can be used to add the `anti-csrf-token` header to the AJAX request. API documentation for `$.ajaxSetup()` can be found here. The function `csrfSafeMethod()` defined below will filter out the safe HTTP methods and only add the header to unsafe HTTP methods.

You can configure jQuery to automatically add the token to all request headers by adopting the following code snippet. This provides a simple and convenient CSRF protection for your AJAX based applications:

```
<script type="text/javascript">
    var csrf_token = $('meta[name="csrf-
token"]').attr('content');

    function csrfSafeMethod(method) {
        // these HTTP methods do not require CSRF
protection
        return (/^(GET|HEAD|OPTIONS)$/.test(method));
    }

    $.ajaxSetup({
        beforeSend: function(xhr, settings) {
            if (!csrfSafeMethod(settings.type) &&
!this.crossDomain) {
                xhr.setRequestHeader("anti-csrf-token",
csrf_token);
            }
        }
    });
</script>
```

This code snippet has been tested with jQuery version 3.3.1.

References

CSRF

- OWASP Cross-Site Request Forgery (CSRF)
- PortSwigger Web Security Academy
- Mozilla Web Security Cheat Sheet

- Common CSRF Prevention Misconceptions
- Robust Defenses for Cross-Site Request Forgery
- For Java: OWASP CSRF Guard or Spring Security
- For PHP and Apache: CSRFProtector Project
- For AngularJS: Cross-Site Request Forgery (XSRF) Protection