Cross-Site Request Forgery Prevention Cheat Sheet

Introduction

A Cross-Site Request Forgery (CSRF) attack occurs when a malicious web site, email, blog, instant message, or program tricks an authenticated user's web browser into performing an unwanted action on a trusted site. If a target user is authenticated to the site, unprotected target sites cannot distinguish between legitimate authorized requests and forged authenticated requests.

Since browser requests automatically include all cookies including session cookies, this attack works unless proper authorization is used, which means that the target site's challenge-response mechanism does not verify the identity and authority of the requester. In effect, CSRF attacks make a target system perform attacker-specified functions via the victim's browser without the victim's knowledge (normally until after the unauthorized actions have been committed).

However, successful CSRF attacks can only exploit the capabilities exposed by the vulnerable application and the user's privileges. Depending on the user's credentials, the attacker can transfer funds, change a password, make an unauthorized purchase, elevate privileges for a target account, or take any action that the user is permitted to do.

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

IMPORTANT: Remember that Cross-Site Scripting (XSS) can defeat all CSRF mitigation techniques!

- **See the OWASP XSS Prevention Cheat Sheet for detailed guidance on how to prevent XSS flaws.**
- **First, check if your framework has built-in CSRF protection and use it**
- **If the 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 the backend**
- **Stateful software should use the synchronizer token pattern**
- **Stateless software should use double submit cookies**

- **If an API-driven site can't use `<form>` tags, consider using custom request headers**
- **Implement at least one mitigation from Defense in Depth Mitigations section**
- **SameSite Cookie Attribute can be used for session cookies** but be careful to NOT set a cookie specifically for a domain. This action introduces a security vulnerability because all subdomains of that domain will share the cookie, and this is particularly an issue if a subdomain has a CNAME to domains not in your control.
- **Consider implementing user interaction based protection for highly sensitive operations**
- **Consider verifying the origin with standard headers**
- **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

Since synchronizer token defenses are built into many frameworks, find out if your framework has CSRF protection available by default before you build a custom token generating system. For example, .NET can use built-in protection to add tokens to CSRF vulnerable resources. If you choose to use this protection, .NET makes you responsible for proper configuration (such as key management and token management).

Synchronizer Token Pattern

CSRF tokens should be generated on the server-side and they should be generated only once per user session or each request. Because the time range for an attacker to exploit the stolen tokens is minimal for per-request tokens, they are more secure than per-session tokens. However, using per-request tokens may result in usability concerns.

For example, the "Back" button browser capability can be hindered by a per-request token as the previous page may contain a token that is no longer valid. In this case, interaction with a previous page will result in a CSRF false positive security event on the server-side. If per-session token implementations occur after the initial generation of a token, the value is stored in the session and is used for each subsequent request until the session expires.

When a client issues a request, the server-side component must verify the existence and validity of the token in that request and compare it to the token found in the user session. The request should be rejected if that token was not found within the request or the value provided does not match the value within the user session. Additional actions such as logging the event as a potential CSRF attack in progress should also be considered.

CSRF tokens should be:

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

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

Transmissing CSRF Tokens in Synchronized Patterns

The CSRF token can be transmitted to the client as part of a response payload, such as a HTML or JSON response, then it can be transmitted back to the server as a hidden field on a form submission or via an AJAX request as a custom header value or part of a JSON payload. A CSRF token should not be transmitted in a cookie for synchronized patterns. A CSRF token must not be leaked in the server logs or in the URL. GET requests can potentially leak CSRF tokens at several locations, such as the browser history, log files, network utilities that log the first line of a 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="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWEwYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1Z
[...]
</form>
```

Since requests with custom headers are automatically subject to the same-origin policy, it is more secure to insert the CSRF token in a custom HTTP request header via JavaScript than adding a CSRF token in the hidden field form parameter.

ALTERNATIVE: Using A Double-Submit Cookie Pattern

If maintaining the state for CSRF token on the server is problematic, you can use an alternative technique known as the Double Submit Cookie pattern. This technique is easy to implement and is stateless. There are different ways to implement this technique,

where the *naive* pattern is the most commonly used variation.

Signed Double-Submit Cookie (RECOMMENDED)

The most secure implementation of the Double Submit Cookie pattern is the *Signed Double-Submit Cookie*, which uses a secret key known only to the server. This ensures that an attacker cannot create and inject their own, known, CSRF token into the victim's authenticated session. The system's tokens should be secured by hashing or encrypting them.

We strongly recommend that you use the Hash-based Message Authentication (HMAC) algorithm because it is less computationally intensive than encrypting and decrypting the cookie. You should also bind the CSRF token with the user's current session to even further enhance security.

EMPLOYING HMAC CSRF TOKENS

To generate HMAC CSRF tokens (with a session-dependent user value), the system must have:

- **A session-dependent value that changes with each login session**. This value should only be valid for the entirety of the users authenticated session. Avoid using static values like the user's email or ID, as they are not secure (1 | 2 | 3). It's worth noting that updating the CSRF token too frequently, such as for each request, is a misconception that assumes it adds substantial security while actually harming the user experience (1). For example, you could choose one, or a combination, of the following session-dependent values:
  - The server-side session ID (e.g. PHP or ASP.NET). This value should never leave the server or be in plain text in the CSRF Token.
  - A random value (e.g. UUID) within a JWT that changes every time a JWT is created.
- **A secret cryptographic key** Not to be confused with the random value from the naive implementation. This value is used to generate the HMAC hash. Ideally, store this key as discussed in the Cryptographic Storage page.
- **A random value for anti-collision purposes**. Generate a random value (preferably cryptographically random) to ensure that consecutive calls within the same second do not produce the same hash (1).

Should Timestamps be Included in CSRF Tokens for Expiration?

It's a common misconception to include timestamps as a value to specify the CSRF token expiration time. A CSRF Token is not an access token. They are used to verify the

authenticity of requests throughout a session, using session information. A new session should generate a new token (1).

PSEUDO-CODE FOR IMPLEMENTING HMAC CSRF TOKENS

Below is an example in pseudo-code that demonstrates the implementation steps described above:

```
// Gather the values
secret = readEnvironmentVariable("CSRF_SECRET") // HMAC secret key
sessionID = session.sessionID // Current authenticated user session
randomValue = cryptographic.randomValue() // Cryptographic random value

// Create the CSRF Token
message = sessionID.length + "!" + sessionID + "!" + randomValue.length +
"!" + randomValue // HMAC message payload
hmac = hmac("SHA256", secret, message) // Generate the HMAC hash
csrfToken = hmac + "." + randomValue // Add the `randomValue` to the HMAC
hash to create the final CSRF token. Avoid using the `message` because it
contains the sessionID in plain text, which the server already stores
separately.

// Store the CSRF Token in a cookie
response.setCookie("csrf_token=" + csrfToken + "; Secure") // Set Cookie
without HttpOnly flag
```

Naive Double-Submit Cookie Pattern (DISCOURAGED)

The *Naive Double-Submit Cookie* method is a scalable and easy-to-implement technique which uses a cryptographically strong random value as a cookie and as a request parameter (even before user authentication). Then the server verifies if the cookie value and request value match. The site must require that every transaction request from the user includes this random value as a hidden form value or inside the request header. If the value matches at server side, the server accepts it as a legitimate request and if they don't, it rejects the request.

Since an attacker is unable to access the cookie value during a cross-site request, they cannot include a matching value in the hidden form value or as a request parameter/ header.

Though the Naive Double-Submit Cookie method is a good initial step to counter CSRF, it still remains vulnerable to certain attacks. This resource provides more information on some vulnerabilities. Thus, we strongly recommend that you use the *Signed Double-Submit Cookie* pattern.

Disallowing simple requests

When a `<form>` tag is used to submit data, it sends a "simple" request that browsers do not designate as "to be preflighted". These "simple" requests introduce risk of CSRF because browsers permit them to be sent to any origin. If your application uses `<form>` tags to submit data anywhere in your client, you will still need to protect them with alternate approaches described in this document such as tokens.

> **Caveat:** Should a browser bug allow custom HTTP headers, or not enforce preflight on non-simple content types, it could compromise your security. Although unlikely, it is prudent to consider this in your threat model. Implementing CSRF tokens adds additional layer of defence and gives developers more control over security of the application.

Disallowing simple content types

For a request to be deemed simple, it must have one of the following content types - `application/x-www-form-urlencoded`, `multipart/form-data` or `text/plain`. Many modern web applications use JSON APIs so would naturally require CORS, however they may accept `text/plain` which would be vulnerable to CSRF. Therefore a simple mitigation is for the server or API to disallow these simple content types.

Employing Custom Request Headers for AJAX/API

Both the synchronizer token and the double-submit cookie are used to prevent forgery of form data, but they can be tricky to implement and degrade usability. Many modern web applications do not use `<form>` tags to submit data. A user-friendly defense that is particularly well suited for AJAX or API endpoints is the use of a **custom request header**. No token is needed for this approach.

In this pattern, the client appends a custom header to requests that require CSRF protection. The header can be any arbitrary key-value pair, as long as it does not conflict with existing headers.

```
X-YOURSITE-CSRF-PROTECTION=1
```

When handling the request, the API checks for the existence of this header. If the header does not exist, the backend rejects the request as potential forgery. This approach has several advantages:

- UI changes are not required

- no server state is introduced to track tokens

This defense relies on the CORS preflight mechanism which sends an `OPTIONS` request to verify CORS compliance with the destination server. All modern browsers designate requests with custom headers as "to be preflighted". When the API verifies that the custom header is there, you know that the request must have been preflighted if it came from a browser.

Custom Headers and CORS

Cookies are not set on cross-origin requests (CORS) by default. To enable cookies on an API, you will set `Access-Control-Allow-Credentials=true`. The browser will reject any response that includes `Access-Control-Allow-Origin=*` if credentials are allowed. To allow CORS requests, but protect against CSRF, you need to make sure the server only allows a few select origins that you definitively control via the `Access-Control-Allow-Origin` header. Any cross-origin request from an allowed domain will be able to set custom headers.

As an example, you might configure your backend to allow CORS with cookies from `http://www.yoursite.com` and `http://mobile.yoursite.com`, so that the only possible preflight responses are:

```
Access-Control-Allow-Origin=http://mobile.yoursite.com
Access-Control-Allow-Credentials=true
```

or

```
Access-Control-Allow-Origin=http://www.yoursite.com
Access-Control-Allow-Credentials=true
```

A less secure configuration would be to configure your backend server to allow CORS from all subdomains of your site using a regular expression. If an attacker is able to take over a subdomain (not uncommon with cloud services) your CORS configuration would allow them to bypass the same origin policy and forge a request with your custom header.

Dealing with Client-Side CSRF Attacks (IMPORTANT)

Client-side CSRF is a new variant of CSRF attacks where the attacker tricks the client-side JavaScript code to send a forged HTTP request to a vulnerable target site by manipulating the program's input parameters. Client-side CSRF originates when the JavaScript program uses attacker-controlled inputs, such as the URL, for the generation

of asynchronous HTTP requests.

Note: These variants of CSRF are particularly important as they can bypass some of the common anti-CSRF countermeasures like token-based mitigations and SameSite cookies. For example, when synchronizer tokens or custom HTTP request headers are used, the JavaScript program will include them in the asynchronous requests. Also, web browsers will include cookies in same-site request contexts initiated by JavaScript programs, circumventing the SameSite cookie policies.

Client-Side vs. Classical CSRF: In the classical CSRF model, the server-side program is the most vulnerable component, because it cannot distinguish whether the incoming authenticated request was performed **intentionally**, also known as the confused deputy problem. In the client-side CSR model, the most vulnerable component is the client-side JavaScript program because an attacker can use it to generate arbitrary asynchronous requests by manipulating the request endpoint and/or its parameters. Client-side CSRF is due to an input validation problem and it reintroduces the confused deputy flaw, that is, the server-side won't, again, be able to distinguish if the request was performed intentionally or not.

For more information about client-side CSRF vulnerabilities, see Sections 2 and 5 of this paper, the CSRF chapter of the SameSite wiki, and this post by the Meta Bug Bounty Program.

Client-side CSRF Example

The following code snippet demonstrates a simple example of a client-side CSRF vulnerability.

```
<script type="text/javascript">
    var csrf_token = document.querySelector("meta[name='csrf-
token']").getAttribute("content");
    function ajaxLoad(){
        // process the URL hash fragment
        let hash_fragment = window.location.hash.slice(1);

        // hash fragment should be of the format: /^(get|post);(.*)$/
        // e.g., https://site.com/index/#post;/profile
        if(hash_fragment.length > 0 && hash_fragment.indexOf(';') > 0 ){

            let params = hash_fragment.match(/^(get|post);(.*)$/);
            if(params && params.length){
                let request_method = params[1];
                let request_endpoint = params[3];

                fetch(request_endpoint, {
                    method: request_method,
```

```
                    headers: {
                        'XSRF-TOKEN': csrf_token,
                        // [...]
                    },
                    // [...]
                }).then(response => { /* [...] */ });
            }
        }
    }
    // trigger the async request on page load
    window.onload = ajaxLoad();
</script>
```

Vulnerability: In this snippet, the program invokes a function `ajaxLoad()` upon the page load, which is responsible for loading various webpage elements. The function reads the value of the URL hash fragment (line 4), and extracts two pieces of information from it (i.e., request method and endpoint) to generate an asynchronous HTTP request (lines 11-13). The vulnerability occurs in lines 15-22, when the JavaScript program uses URL fragments to obtain the server-side endpoint for the asynchronous HTTP request (line 15) and the request method. However, both inputs can be controlled by web attackers, who can pick the value of their choosing, and craft a malicious URL containing the attack payload.

Attack: Usually, attackers share a malicious URL with the victim (through elements such as spear-phishing emails) and because the malicious URL appears to be from an honest, reputable (but vulnerable) website, the user often clicks on it. Alternatively, the attackers can create an attack page to abuse browser APIs (e.g., the `window.open()` API) and trick the vulnerable JavaScript of the target page to send the HTTP request, which closely resembles the attack model of the classical CSRF attacks.

For more examples of client-side CSRF, see this post by the Meta Bug Bounty Program and this USENIX Security paper.

Client-side CSRF Mitigation Techniques

Independent Requests: Client-side CSRF can be prevented when asynchronous requests cannot be generated via attacker controllable inputs, such as the URL, window name, document referrer, and postMessages, to name only a few examples.

Input Validation: Achieving complete isolation between inputs and request parameters may not always be possible depending on the context and functionality. In these cases, input validation checks has to be implemented. These checks should strictly assess the format and choice of the values of the request parameters and decide whether they can only be used in non-state-changing operations (e.g., only allow GET requests and

endpoints starting with a predefined prefix).

Predefined Request Data: Another mitigation technique is to store a list of predefined, safe request data in the JavaScript code (e.g., combinations of endpoints, request methods and other parameters that are safe to be replayed). The program can then use a switch parameter in the URL fragment to decide which entry of the list should each JavaScript function use.

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, if a GitHub-like website uses the Strict value, a logged-in GitHub user who tries to follow a link to a private GitHub project posted on a corporate discussion forum or email, the user will not be able to access the project because GitHub will not receive a session cookie. Since a bank website would not allow any transactional pages to be linked from external sites, so the Strict flag would be most appropriate for banks.

If a website wants to maintain a user's logged-in session after the user arrives from an external link, SameSite's default Lax value provides a reasonable balance between security and usability. If the GitHub scenario above uses a Lax value instead, 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 have top-level navigations and use 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 track the browsers implementing it and know how the attribute is used, 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 a CSRF Token. Instead, it should co-exist with that token to protect the user in a more robust way.

Using Standard Headers to Verify Origin

There are two steps to this mitigation method, both of which examine an HTTP request header value:

1. Determine the origin that the request is coming from (source origin). Can be done via Origin or Referer headers.
2. Determining the origin that 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 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 ORIGIN HEADER IS NOT PRESENT

If the Origin header is not present, verify that 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

Generally, it's not always easy to determine the target origin. You are not always able to simply grab the target origin (i.e., its hostname and port `#`) from the URL in the request, because the application server is frequently sitting behind one or more proxies. This means that the original URL can be different from the URL the app server actually receives. However, 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:** Since it is your application, you can find its target origin and set that value in some server configuration entry. This would be the most secure approach as its 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 provide your instances the ability to grab the 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 want your application to 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 is meant 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 possibility that the proxy will alter the host header, you can use another header called X-Forwarded-Host 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 the value in X-Forwarded-Host is likely to be the target origin value that you need to compare to the source origin in the Origin or Referer header.

Using this header value for mitigation will work properly when origin or referrer headers are present in the requests. Though these headers are included the **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).

Use cases where X-Forward-Host is not employed:

- 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.

Using Cookies with Host Prefixes to Identify Origins

While the `SameSite` and `Secure` attributes mentioned earlier restrict the sending of already set cookies and `HttpOnly` restricts the reading of a set cookie, an attacker may still try to inject or overwrite otherwise secured cookies (cf. session fixation attacks). Using `Cookie Prefixes` for cookies with CSRF tokens extends security protections against this kind of attacks as well. If cookies have `__Host-` prefixes e.g. `Set-Cookie: __Host-token=RANDOM; path=/; Secure` then each cookie:

- Cannot be (over)written from another subdomain and
- cannot have a `Domain` attribute.
- Must have the path of `/`.

- Must be marked as Secure (i.e, cannot be sent over unencrypted HTTP).

In addition to the `__Host-` prefix, the weaker `__Secure-` prefix is also supported by browser vendors. It relaxes the restrictions on domain overwrites, i.e., they

- Can have `Domain` attributes and
- can be overwritten by subdomains.
- Can have a `Path` other than `/`.

This relaxed variant can be used as an alternative to the "domain locked" `__Host-` prefix, if authenticated users would need to visit different (sub-)domains. In all other cases, using the `__Host-` prefix in addition to the `SameSite` attribute is recommended.

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

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

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 mechanisms
- One-time Tokens

Do NOT use CAPTCHA because it is specifically designed to protect against bots. It is possible, and still valid in some implementations of CAPTCHA, to obtain proof of human interaction/presence from a different user session. Although this makes the CSRF exploit more complex, it does not protect against it.

While these are very strong CSRF defenses, 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 changes, money transfers, etc.), alongside the other defences discussed in this cheat sheet.

Possible CSRF Vulnerabilities in Login Forms

Most developers tend to ignore CSRF vulnerabilities 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. Login CSRF can also be mitigated by including a custom request headers in AJAX request as described above.

REFERENCE: Sample JEE Filter Demonstrating CSRF Protection

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 this is only 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: Automatically Including CSRF Tokens as an AJAX Request Header

The following guidance for JavaScript by default considers **GET**, **HEAD** and **OPTIONS** methods as 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 a 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 the CSRF token 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 you to 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 in Related Cheat Sheets

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