[Cross-Site Scripting](https://www.owasp.org/index.php/Cross-Site_Scripting" \t "_blank) is not necessary for CSRF to work. However, any cross-site scripting vulnerability can be used to defeat all CSRF mitigation techniques available in the market today (except mitigation techniques that involve user interaction and described later in this cheatsheet). This is because an XSS payload can simply read any page on the site using an XMLHttpRequest (direct DOM access can be done, if on same page) and obtain the generated token from the response, and include that token with a forged request. This technique is exactly how the [MySpace (Samy) worm](https://en.wikipedia.org/wiki/Samy_%28computer_worm%29" \t "_blank) defeated MySpace's anti-CSRF defenses in 2005, which enabled the worm to propagate.

It is imperative that no XSS vulnerabilities are present to ensure that CSRF defenses can't be circumvented. Please see the OWASP [XSS Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html) for detailed guidance on how to prevent XSS flaws.

**Resources that need to be protected from CSRF vulnerability**

The following list assumes that you are not violating [RFC2616](http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.1.1), section 9.1.1, by using GET requests for state changing operations.

**Note:** If for any reason you violate, you would also need to protect those resources, which is mostly achieved with default form tag [GET method], href, and src attributes.

* Form tags with POST
* Ajax/XHR calls

**CSRF Defense Recommendations Summary**

We recommend token based CSRF defense (either stateful/stateless) as a primary defense to mitigate CSRF in your applications. Only for highly sensitive operations, we also recommend a user interaction based protection (either re-authentication/one-time token, detailed in section 6.5) along with token based mitigation.

As a defense-in-depth measure, consider implementing one mitigation from Defense in Depth Mitigations section (you can choose the mitigation that fits your ecosystem considering the issues mentioned under them). These defense-in-depth mitigation techniques are not recommended to be used by themselves (without token based mitigation) for mitigating CSRF in your applications.

**Primary Defense Technique**

**Token Based Mitigation**

This defense is one of the most popular and recommended methods to mitigate CSRF. It can be achieved either with state (synchronizer token pattern) or stateless (encrypted/hash based token pattern). See section 4.3 on how to mitigate login CSRF in your applications. For all the mitigations, it is implicit that general security principles should be adhered

* Strong encryption/HMAC functions should be adhered to.

**Note:** You can select any algorithm per your organizational needs. We recommend AES256-GCM for encryption and SHA256/512 for HMAC.

* Strict key rotation and token lifetime policies should be maintained. Policies can be set according to your organizational needs. Generic key management guidance from OWASP can be found [here](https://cheatsheetseries.owasp.org/cheatsheets/Key_Management_Cheat_Sheet.html).

**Synchronizer Token Pattern**

Any state changing operation requires a secure random token (e.g., CSRF token) to prevent CSRF attacks. A CSRF token should be:

* unique per user session;
* large random value;
* generated by a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG).

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](https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#disclosure-of-token-in-url). The server must reject the requested action if the CSRF token fails validation.

Inserting the CSRF token in the HTTP request header via JavaScript is considered more secure than adding the token in the hidden field form parameter. In this situation, even if the CSRF token is weak, predictable or leaked but still an attacker cannot forge the POST request directly by setting the custom request header through XMLHttpRequest. As per the [MDN documentation](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#Preflighted_requests) when the attacker tries to set any custom header through any XMLHttpRequest, the browser sends the OPTIONS (pre-flight) request. Moreover, when the attacker tries to spoof the header through flash, the browser initiates the GET request for crossdomain.xml. In both cases, the browser prevents the forged request to be sent.

In order to facilitate a "transparent but visible" CSRF solution, developers are encouraged to adopt a pattern similar to [Synchronizer Token Pattern](http://www.corej2eepatterns.com/Design/PresoDesign.htm) (The original intention of this synchronizer token pattern was to detect duplicate submissions in forms). The synchronizer token pattern requires the generation of random "challenge" tokens that are associated with the user's current session. These challenge tokens are then inserted within the HTML forms and calls associated with sensitive server-side operations. It is the responsibility of the server application to verify the existence and correctness of this token. By including a challenge token with each request, the developer has a strong control to verify that the user actually intended to submit the desired requests. Inclusion of a required security token in HTTP requests associated with sensitive business functions helps mitigate CSRF attacks as successful exploitation assumes the attacker knows the randomly generated token for the target victim's session.

**Note:** These tokens aren't like cookies that are automatically sent with forged requests made from your browser from the attacker website.

This is analogous to the attacker being able to guess the target victim's session identifier.

The following describes a general approach to incorporate challenge tokens within the request.

When a Web application formulates a request, the application should include a hidden input parameter with a common name such as "CSRFToken" (for forms)/ as header/parameter value for Ajax calls. The value of this token must be randomly generated such that it cannot be guessed by an attacker. Consider leveraging the java.security.SecureRandom class for Java applications to generate a sufficiently long random token. Alternative generation algorithms include the use of 256-bit BASE64 encoded hashes. Developers that choose this generation algorithm must make sure that there is randomness and uniqueness utilized in the data that is hashed to generate the random token.

<form action="/transfer.do" method="post">

<input type="hidden" name="CSRFToken"

value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWEwYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZMGYwMGEwOA==">

...

</form>

In general, developers need only generate this token once for the current session. After initial generation of this 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 end-user, 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, and the event logged as a potential CSRF attack in progress.

To further enhance the security of this proposed design, consider randomizing the CSRF token parameter name and/or value for each request. Implementing this approach results in the generation of per-request tokens as opposed to per-session tokens. This is 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. Few applications that need high security typically implement this approach (such as banks). You have to check what suits your needs. Regardless of the approach taken, developers are encouraged to protect the CSRF token the same way they protect authenticated session identifiers, such as the use of TLS.

**Existing Synchronizer Implementations**

Synchronizer token defenses have been built into many frameworks, so we strongly recommend using them first when they are available. External components that add CSRF defenses to existing applications are also recommended. OWASP has the following:

* For Java: OWASP [CSRF Guard](https://www.owasp.org/index.php/CSRF_Guard)
* For PHP and Apache: [CSRFProtector Project](https://www.owasp.org/index.php/CSRFProtector_Project" \t "_blank)

**Disclosure of Token in URL**

Some implementations of synchronizer tokens include the challenge token in GET (URL) requests as well as POST requests. This is often implemented as a result of sensitive server-side operations being invoked as a result of embedded links in the page or other general design patterns. While this control does help mitigate the risk of CSRF attacks, the unique per-session token is being exposed for GET requests. CSRF tokens in GET requests are potentially leaked at several locations: browser history, log files, network appliances that make a point to log the first line of an HTTP request, and Referer headers if the protected site links to an external site. In the latter case (leaked CSRF token due to the Referer header being parsed by a linked site), it is trivially easy for the linked site to launch a CSRF attack on the protected site, and they will be able to target this attack very effectively, since the Referer header tells them the site as well as the CSRF token.

The ideal solution is to only include the CSRF token in POST requests and modify server-side actions that have state changing affect to only respond to POST requests. This is in fact what the RFC 2616 requires for GET requests. If sensitive server-side actions are guaranteed to only ever respond to POST requests, then there is no need to include the token in GET requests.

**Encryption based Token Pattern**

The Encrypted Token Pattern leverages an encryption, rather than comparison method of Token-validation. It is most suitable for applications that do not want to maintain any state at server side.

Server generates a token comprised of the user's session ID and a timestamp (to prevent replay attacks) using a unique key available only on the server (AES256-with GCM mode/GCM-SIV is recommended. Usage of ECB mode is strictly not recommended. If you would like to use any other block cipher mode of operation, refer [here](https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html) and [here](https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation) for more information). This token is returned to the client and embedded in a hidden field for forms, in the request-header/parameter for AJAX requests. On receipt of this request, the server reads and decrypts the token value with the same key used to create the token. Inability to correctly decrypt suggest an intrusion attempt (recommend to block and log the attack for incident response purposes). Once decrypted, the users sessionId and timestamp contained within the token are validated; session-id is compared against the currently logged in user, and the timestamp is compared against the current time to verify that its not beyond the defined token expiry time. If session-id matches and the timestamp is under the defined token expiry time, request can be allowed.

Refer [here](https://cheatsheetseries.owasp.org/cheatsheets/Key_Management_Cheat_Sheet.html#key-management-lifecycle-best-practices) to learn best practices about managing your encryption key.

**HMAC Based Token Pattern**

HMAC Based Token Pattern mitigation is also achieved without maintaining any state at the server. [HMAC](https://en.wikipedia.org/wiki/HMAC) based CSRF protection works similar to encryption based CSRF protection with a couple of minor differences

1. Use strong HMAC function (SHA256 or stronger algorithms are recommended) instead of an encryption function to generate the token
2. Token is HMAC+timestamp

Below are the steps for the proper implementation of the HMAC based CSRF protection:

1. Generate the token\ Using key K, generate HMAC(usersessionID+timestamp) and append the same timestamp value to it which results in your CSRF token.
2. Include the token (*i.e.* HMAC+timestamp)\ Include token in a hidden field for forms and in the request-header field/request body parameter for AJAX requests.
3. Validating the token\ When the request is received at the server, re-generate the token with same key K (parameters are sessionID from the request and timestamp in the received token). If the HMAC in the received token and the one generated in this step match, verify if timestamp received is less than defined token expiry time. If both of them are success, then request is treated as legitimate and can be allowed. If not, block the request and log the attack for incident response purposes.

Refer [here](https://cheatsheetseries.owasp.org/cheatsheets/Key_Management_Cheat_Sheet.html#key-management-lifecycle-best-practices) to learn best practices about managing your HMAC key.

**Auto CSRF Mitigation Techniques**

Though the technique of mitigating tokens is widely used (stateful with synchronizer token and stateless with encrypted/HMAC token), the major problem associated with these techniques is the human tendency to forget things at times. If a developer forgets to add the token to any state changing operation, they are making the application vulnerable to CSRF. To avoid this, you can try to automate the process of adding tokens to CSRF vulnerable resources (mentioned earlier in this document). You can achieve this by doing the following:

* Write wrappers (that would auto add tokens when used) around default form tags/ajax calls and educate your developers to use those wrappers instead of standard tags. Though this approach is better than depending purely on developers to add tokens, it still is vulnerable to the issue of human tendency to forget things. [Spring Security](https://docs.spring.io/spring-security/site/docs/3.2.0.CI-SNAPSHOT/reference/html/csrf.html) uses this technique to add CSRF tokens by default when a custom <form:form> tag is used, you can opt to use after verifying that its enabled and properly configured in the Spring Security version you are using.
* Write a hook (that would capture the traffic and add tokens to CSRF vulnerable resources before rendering to customers) in your organizational web rendering frameworks. Because it is hard to analyze when a particular response is doing any state change (and thus needing a token), you might want to include tokens in all CSRF vulnerable resources (ex: include tokens in all POST responses). This is one recommended approach, but you need to consider the performance costs it might incur.
* Get the tokens automatically added on the client side when the page is being rendered in user's browser, with help of a client side script (this approach is used by [CSRF Guard](https://www.owasp.org/index.php/CSRF_Guard)). You need to consider any possible JavaScript hijacking attacks.

We recommend researching 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](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-2.1) 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.

**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. That assumption is false. CSRF vulnerability can still occur on login forms where the user is not authenticated, but the impact/risk view for it is quite different from the impact/risk view of a general CSRF vulnerability (when a user is authenticated).

With a CSRF vulnerability on login form, an attacker can make a victim login as them and learn behavior from their searches. For more information about login CSRF and other risks, see section 3 of [this](https://seclab.stanford.edu/websec/csrf/csrf.pdf) 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. Pre-sessions can be transitioned to real sessions once the user is authenticated. This technique is described in [Robust Defenses for Cross-Site Request Forgery section 4.1](https://seclab.stanford.edu/websec/csrf/csrf.pdf).

If sub-domains under your master domain are treated as not trusty in your threat model, it is difficult to mitigate login CSRF. A strict subdomain and path level referrer header (because most login pages are served on HTTPS - no stripping of referrer - and are also linked from home pages) validation (detailed in section 6.1) can be used in these cases for mitigating CSRF on login forms to an extent.

**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](https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis-02#section-5.3.7). 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](https://tools.ietf.org/html/rfc7231#section-4.2.1) HTTP methods.

For more details on the SameSite values, check the following [section](https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis-02#section-5.3.7.1) from the [rfc](https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis-02" \t "_blank).

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](https://caniuse.com/#feat=same-site-cookie-attribute). Note that Chrome has [announced](https://blog.chromium.org/2019/10/developers-get-ready-for-new.html) 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](https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis-02#section-5.3.7.1). 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**

This defense technique is specifically proposed in section 5.0 of [Robust Defenses for Cross-Site Request Forgery](https://seclab.stanford.edu/websec/csrf/csrf.pdf). This paper proposes the first creation of the Origin header and its use as a CSRF defense mechanism.

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 and/or referer header.
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 in an XSS) as they fall under [forbidden headers](https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name) list (i.e., only browsers 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 site.com make sure site.com.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 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 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 stackoverflow [here](https://stackoverflow.com/questions/20784209/internet-explorer-11-does-not-add-the-origin-header-on-a-cors-request) and [here](https://github.com/silverstripe/silverstripe-graphql/issues/118).
* In an instance following a [302 redirect cross-origin](https://stackoverflow.com/questions/22397072/are-there-any-browsers-that-set-the-origin-header-to-null-for-privacy-sensitiv), 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](https://wiki.mozilla.org/Security/Origin#Privacy-Sensitive_Contexts) where Origin is set to "null" For example, see the following [here](https://www.google.com/search?q=origin+header+sent+null+value+site%3Astackoverflow.com&oq=origin+header+sent+null+value+site%3Astackoverflow.com).
* 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](https://stackoverflow.com/questions/6880659/in-what-cases-will-http-referer-be-empty), [2](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referer), [3](https://en.wikipedia.org/wiki/HTTP_referer#Referer_hiding), [4](https://seclab.stanford.edu/websec/csrf/csrf.pdf) and [5](https://www.google.com/search?q=referrer+header+sent+null+value+site:stackoverflow.com)). 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%](http://homakov.blogspot.com/2012/04/playing-with-referer-origin-disquscom.html)) and no enterprise would want to lose even this minor % of traffic. One of the popular technique used across 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](http://homakov.blogspot.com/2012/04/playing-with-referer-origin-disquscom.html). 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 minor effort involved in deploying it

**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.

**This technique works as long as you are sure that your subdomains are fully secured and only accept HTTPS connections** The problem of "trusting of sub domains and proper configuration of whole site in general to accept HTTPS connections only". The [Blackhat talk](https://media.blackhat.com/eu-13/briefings/Lundeen/bh-eu-13-deputies-still-confused-lundeen-wp.pdf) by Rich Lundeen references these drawbacks.

"*With double submit, if an attacker can write a cookie they can obviously defeat the protection. And again, writing cookies is significantly easier then reading them. The fact that cookies can be written is difficult for many people to understand. After all, doesn't the same origin policy specify that one domain cannot access cookies from another domain? However, there are two common scenarios where writing cookies across domains is possible:*

*a) While it's true that hellokitty.marketing.example.com cannot read cookies or access the DOM from secure.example.com because of the same origin policy, hellokitty.marketing.example.com can write cookies to the parent domain (example.com), and these cookies are then consumed by secure.example.com (secure.example.com has no good way to distinguish which site set the cookie). Additionally, there are methods of forcing secure.example.com to always accept your cookie first. What this means is that XSS in hellokitty.marketing.example.com is able to overwrite cookies in secure.example.com.*

*b) If an attacker is in the middle, they can usually force a request to the same domain over HTTP. If an application is hosted at https://secure.example.com, even if the cookies are set with the secure flag, a man in the middle can force connections to http://secure.example.com and set (overwrite) any arbitrary cookies (even though the secure flag prevents the attacker from reading those cookies). Even if the HSTS header is set on the server and the browser visiting the site supports HSTS (this would prevent a man in the middle from forcing plaintext HTTP requests) unless the HSTS header is set in a way that includes all subdomains, a man in the middle can simply force a request to a separate subdomain and overwrite cookies similar to 1. In other words, as long as http://hellokitty.marketing.example.com doesn't force https, then an attacker can overwrite cookies on any example.com subdomain.*"

Scenarios a and b mentioned above are possible only if the CSRF token is not derived/tied to the session in which case an attacker can overwrite the token in the parent domain cookie with XSS in child domain. A variant of how this can be mitigated by linking token and session/auth cookie is explained below.

Including the token in an encrypted cookie - other than the authentication cookie (since they are often shared within subdomains) - and then at the server side matching it (after decrypting the encrypted cookie) with the token in hidden form field or parameter/header for ajax calls mitigates both the issues mentioned above. 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 hash the token with a secret salt 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 salted-hash is used, an attacker won't be able to recreate the cookie value from the plain token without knowledge of the server secrets.

**Note about Triple Submit Cookie**

This mitigation is proposed by John Wilander in 2012 at OWASP Appsec Research. This technique adds an additional step to double submit cookie approach by verifying if the request contains two cookies with same name (please note, attacker need to write an additional cookie to bypass double submit cookie mitigation). Though it mitigates the issues discussed in bypass of double submit cookie, it introduces new problems such as cookie jar overflow (in-details and more issue details [here](https://media.blackhat.com/eu-13/briefings/Lundeen/bh-eu-13-deputies-still-confused-lundeen-wp.pdf) and [here](https://webstersprodigy.net/2012/08/03/analysis-of-john-wilanders-triple-submit-cookies/)

**Use of Custom Request Headers**

Adding CSRF tokens, a double submit cookie and value, 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/XHR endpoints is the use of a custom request header. This defense relies on the [same-origin policy (SOP)](https://en.wikipedia.org/wiki/Same-origin_policy) 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.

A particularly attractive custom header and value to use is "X-Requested-With: XMLHttpRequest" because most JavaScript libraries already add this header to requests they generate by default. However, some do not. For example, AngularJS used to, but does not anymore. For more information, see [their rationale](https://github.com/angular/angular.js/commit/3a75b1124d062f64093a90b26630938558909e8d) and how to add it back.

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 defense technique is specifically discussed in section 4.3 of [Robust Defenses for Cross-Site Request Forgery](https://seclab.stanford.edu/websec/csrf/csrf.pdf). However, bypasses of this defense using Flash were documented as early as 2008 and again as recently as 2015 by Mathias Karlsson to [exploit a CSRF flaw in Vimeo](https://hackerone.com/reports/44146). Riyaz Walikar from [Appsecco Labs](https://appsecco.com/" \t "_blank) was able to exploit this even in early 2019 [using older versions of Firefox](https://blog.appsecco.com/exploiting-csrf-on-json-endpoints-with-flash-and-redirects-681d4ad6b31b) and also in some recent versions of Chrome (with a bit of [user involvement](https://github.com/sp1d3r/swf_json_csrf/blob/master/read.html)). Since we cannot the control the versions of browser end user uses nor depend on it for their security, this technique is not recommended as a primary defense in measure.

This technique obviously works for Ajax calls and you have to still need 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).

**Note about Content-Type Header Validation**

Do not use Content-Type Header as a CSRF mitigation. For more information why, see [Common CSRF Prevention Misconceptions](https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2017/september/common-csrf-prevention-misconceptions/)

**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 (password or stronger)
* One-time Token
* CAPTCHA

While these are a very strong CSRF defense, it does create a huge impact on the user experience. For applications that are in need of high security for some operations (password change, money transfer etc.), these techniques should be used along with token based mitigation. Please note that tokens by themselves can mitigate CSRF, developers should use these techniques only to achieve additional security for their high sensitive operations.

**Implementation 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](https://github.com/aramrami/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 standard authentication or authorization checks before checking for CSRF.

Source is also located [here](https://github.com/righettod/poc-csrf) and provides a runnable POC.

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import javax.servlet.annotation.WebFilter;

import javax.servlet.http.Cookie;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.http.HttpServletResponseWrapper;

import javax.xml.bind.DatatypeConverter;

import java.io.IOException;

import java.net.MalformedURLException;

import java.net.URL;

import java.security.SecureRandom;

import java.util.Arrays;

/\*\*

\* Filter in charge of validating each incoming HTTP request about Headers

\* and CSRF token.

\* It is called for all requests to backend destination.

\*

\* We use the approach in which:

\* - The CSRF token is changed after each valid HTTP exchange

\* - The custom Header name for the CSRF token transmission is fixed

\* - A CSRF token is associated to a backend service URI in order to enable

\* the support for multiple parallel Ajax request from the same application

\* - The CSRF cookie name is the backend service name prefixed with a fixed prefix

\*

\* Here for the POC we show the "access denied" reason in the response but in

\* production code only return a generic message !

\*

\* @see "https://wiki.mozilla.org/Security/Origin"

\* @see "https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie"

\* @see "https://chloe.re/2016/04/13/goodbye-csrf-samesite-to-the-rescue/"

\*/

@WebFilter("/backend/\*")

public class CSRFValidationFilter implements Filter {

/\*\*

\* JVM param name used to define the target origin

\*/

public static final String TARGET\_ORIGIN\_JVM\_PARAM\_NAME = "target.origin";

/\*\*

\* Name of the custom HTTP header used to transmit the CSRF token and also to prefix

\* the CSRF cookie for the expected backend service

\*/

private static final String CSRF\_TOKEN\_NAME = "X-TOKEN";

/\*\*

\* Logger

\*/

private static final Logger LOG = LoggerFactory.getLogger(CSRFValidationFilter.class);

/\*\*

\* Application expected deployment domain: named "Target Origin" in OWASP CSRF article

\*/

private URL targetOrigin;

/\*\*\*

\* Secure generator

\*/

private final SecureRandom secureRandom = new SecureRandom();

/\*\*

\* {@inheritDoc}

\*/

@Override

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

throws IOException, ServletException {

HttpServletRequest httpReq = (HttpServletRequest) request;

HttpServletResponse httpResp = (HttpServletResponse) response;

String accessDeniedReason;

/\* STEP 1: Verifying Same Origin with Standard Headers \*/

//Try to get the source from the "Origin" header

String source = httpReq.getHeader("Origin");

if (this.isBlank(source)) {

//If empty then fallback on "Referer" header

source = httpReq.getHeader("Referer");

//If this one is empty too then we trace the event and we block the request

//(recommendation of the article)...

if (this.isBlank(source)) {

accessDeniedReason = "CSRFValidationFilter: ORIGIN and REFERER request" +

"headers are both absent/empty so we block the request !";

LOG.warn(accessDeniedReason);

httpResp.sendError(HttpServletResponse.SC\_FORBIDDEN, accessDeniedReason);

return;

}

}

//Compare the source against the expected target origin

URL sourceURL = new URL(source);

if (!this.targetOrigin.getProtocol().equals(sourceURL.getProtocol()) ||

!this.targetOrigin.getHost().equals(sourceURL.getHost())

|| this.targetOrigin.getPort() != sourceURL.getPort()) {

//One the part do not match so we trace the event and we block the request

accessDeniedReason = String.format("CSRFValidationFilter: Protocol/Host/Port " +

"do not fully matches so we block the request! (%s != %s) ",

this.targetOrigin, sourceURL);

LOG.warn(accessDeniedReason);

httpResp.sendError(HttpServletResponse.SC\_FORBIDDEN, accessDeniedReason);

return;

}

/\* STEP 2: Verifying CSRF token using "Double Submit Cookie" approach \*/

//If CSRF token cookie is absent from the request then we provide one

//in response but we stop the process at this stage.

//Using this way we implement the first providing of token

Cookie tokenCookie = null;

if (httpReq.getCookies() != null) {

String csrfCookieExpectedName = this.determineCookieName(httpReq);

tokenCookie = Arrays.stream(httpReq.getCookies()).filter(c

-> c.getName().equals(csrfCookieExpectedName)).findFirst().orElse(null);

}

if (tokenCookie == null || this.isBlank(tokenCookie.getValue())) {

LOG.info("CSRFValidationFilter: CSRF cookie absent or value" +

" is null/empty so we provide one and return an HTTP NO\_CONTENT response !");

//Add the CSRF token cookie and header

this.addTokenCookieAndHeader(httpReq, httpResp);

//Set response state to "204 No Content" in order to allow the requester to

//clearly identify an initial response providing the initial CSRF token

httpResp.setStatus(HttpServletResponse.SC\_NO\_CONTENT);

} else {

//If the cookie is present then we pass to validation phase

//Get token from the custom HTTP header (part under control of the requester)

String tokenFromHeader = httpReq.getHeader(CSRF\_TOKEN\_NAME);

//If empty then we trace the event and we block the request

if (this.isBlank(tokenFromHeader)) {

accessDeniedReason = "CSRFValidationFilter: Token provided via HTTP Header"+

" is absent/empty so we block the request !";

LOG.warn(accessDeniedReason);

httpResp.sendError(HttpServletResponse.SC\_FORBIDDEN, accessDeniedReason);

} else if (!tokenFromHeader.equals(tokenCookie.getValue())) {

//Verify that token from header and one from cookie are the same

//Here is not the case so we trace the event and we block the request

accessDeniedReason = "CSRFValidationFilter: Token provided via HTTP Header"+

"and via Cookie are not equals so we block the request !";

LOG.warn(accessDeniedReason);

httpResp.sendError(HttpServletResponse.SC\_FORBIDDEN, accessDeniedReason);

} else {

//Verify that token from header and one from cookie matches

//Here is the case so we let the request reach the target component

//(ServiceServlet, jsp...) and add a new token when we get back the bucket

HttpServletResponseWrapper httpRespWrapper =

new HttpServletResponseWrapper(httpResp);

chain.doFilter(request, httpRespWrapper);

//Add the CSRF token cookie and header

this.addTokenCookieAndHeader(httpReq, httpRespWrapper);

}

}

}

/\*\*

\* {@inheritDoc}

\*/

@Override

public void init(FilterConfig filterConfig) throws ServletException {

//To easier the configuration, we load the target expected origin from

//an JVM property

//Reconfiguration only require an application restart that is generally acceptable

try {

this.targetOrigin = new URL(System.getProperty(TARGET\_ORIGIN\_JVM\_PARAM\_NAME));

} catch (MalformedURLException e) {

LOG.error("Cannot init the filter !", e);

throw new ServletException(e);

}

LOG.info("CSRFValidationFilter: Filter init, set expected target origin to '{}'.",

this.targetOrigin);

}

/\*\*

\* {@inheritDoc}

\*/

@Override

public void destroy() {

LOG.info("CSRFValidationFilter: Filter shutdown");

}

/\*\*

\* Check if a string is null or empty (including containing only spaces)

\*

\* @param s Source string

\* @return TRUE if source string is null or empty (including containing only spaces)

\*/

private boolean isBlank(String s) {

return s == null || s.trim().isEmpty();

}

/\*\*

\* Generate a new CSRF token

\*

\* @return The token a string

\*/

private String generateToken() {

byte[] buffer = new byte[50];

this.secureRandom.nextBytes(buffer);

return DatatypeConverter.printHexBinary(buffer);

}

/\*\*

\* Determine the name of the CSRF cookie for the targeted backend service

\*

\* @param httpRequest Source HTTP request

\* @return The name of the cookie as a string

\*/

private String determineCookieName(HttpServletRequest httpRequest) {

String backendServiceName = httpRequest.getRequestURI().replaceAll("/", "-");

return CSRF\_TOKEN\_NAME + "-" + backendServiceName;

}

/\*\*

\* Add the CSRF token cookie and header to the provided HTTP response object

\*

\* @param httpRequest Source HTTP request

\* @param httpResponse HTTP response object to update

\*/

private void addTokenCookieAndHeader(HttpServletRequest httpRequest,

HttpServletResponse httpResponse) {

//Get new token

String token = this.generateToken();

//Add cookie manually because the current Cookie class implementation

//do not support the "SameSite" attribute

//We let the adding of the "Secure" cookie attribute to the reverse proxy rewriting...

//Here we lock the cookie from JS access and we use the SameSite new attribute protection

String cookieSpec = String.format("%s=%s; Path=%s; HttpOnly; SameSite=Strict",

this.determineCookieName(httpRequest), token, httpRequest.getRequestURI());

httpResponse.addHeader("Set-Cookie", cookieSpec);

//Add cookie header to give access to the token to the JS code

httpResponse.setHeader(CSRF\_TOKEN\_NAME, token);

}

}

**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.

**POST**, **PUT**, **PATCH**, **DELETE** and **TRACE** verbs, 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 JS variable / 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 [$httpProvider](https://docs.angularjs.org/api/ng/provider/$httpProvider#defaults).

<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](https://github.com/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>