Practical Security - Double Submit Cookie Method

# Overview

A method of using JSON Web Tokens (JWTs) in HttpOnly cookies to provide authentication for APIs, while mitigating Cross-Site Request Forgery attacks.

# The Challenge

## JWTs

JWTs are an authentication mechanism built Base64 encoded JSON object, which is cryptographically signed by the server to detect tampering. Generally, the JSON will contain a selection of claims, commonly: the issuer, user identifiers, and an expiration date. When a user logs into a site, a JWT is generated and returned to the user. When a new request is made the JWT is transferred to the server and the claims provided in the token allow a server to authenticate a user for specific resources.

The challenge with authenticating using JWTs, is both were to store the token, and how to send it, while reducing the ability for an attacker to expolit the authentication and the impact of an exploits.

## HTTPS

When performing client authentication and sending JWTs between the client and server, the connection should be secure. JWTs are not normally encrypted and if intercepted by a third party, could comprimise the site authentication system. Therefore, always use HTTPS as standard when a site requires any form of authentication.

## Where to Store JWTs

There are two places to store JWTs on the client machine, in the HTML5 local session storage and in a cookie.

## Local Storage

Local session storage is easy to use, however it is vunerable to XSS attacks, and since it has no data protection, it can be accessed by other websites JavaScript easily. The advantage of local storage for storing JWTs, is that it is not vunerable to XSRF attacks, since site JavaScript is required to access it.

## Cookies

Cookies are set by the server, so storing a JWT in a HttpOnly cookie is easy to perform and due to the design of cookies, it will automatically get sent with each request to the server, allowing for easy authentication on restricted resources, without any client side processing.

Another design feature of HttpOnly cookies is that they are only accessible by the server which created them, in the headers of the request. Therefore, they are generally except from XSS attacks, excluding some advanced attacks such as XST (<https://stackoverflow.com/questions/228138/is-it-possible-for-a-xss-attack-to-obtain-httponly-cookies#:~:text=Packet%20sniffing%20can%20read%20the,not%20fall%20under%20the%20XSS.&text=JavaScript%20can%20modify%20the%20HTML,you%20are%20safe%20against%20XSS>.) This design makes HttpOnly cookies much more suitable for storing senstive data than local storage.

The main issue with using cookies to store the token they are vunerable to XSRF attacks. XSRF attacks are where a malicious website tricks a user into sending a request to the server (through a fake img tag, form, or similar), and since HttpOnly cookies are always sent with requests to their server, the malicious request will be authenticated. If an attacker knows the architecture of the server, damaging actions can be performed, such as deleting or changing user and company information.

# The Solution

## General

Since HttpOnly cookies have much better protection than HTML local storage, they are the obvious choice for JWT storage. To mitigate XSRF attacks a process called the double submit cookie method is used.

## The Double Submit Cookie Method

The double submit cookie method requires giving the client two cookies:

* JWT HttpOnly cookie (validates user authentication)
* X-XSRF-TOKEN cookie (ensures request is from host website)

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 HMAC signed 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 request parameter or header. If both of them match at server side, the server accepts it as legitimate request and if they don't, it would reject the request.

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

# Code It

## Enable HTTPS

Enable in application or proxy as standard

## Server Side Functions

### Install Dependencies

Add cookie-parser to create and sign cookies:

yarn add cookie-parser

Add csurf to generate CSRF tokens for client submital on mutating request:

yarn add csurf

### Setup Cookie-Parser

Pass through site secret for signing cookies on initalization:

const cookieParser = require('cookie-parser')

app.use(cookieParser(process.env.SITE\_SECRET))

### Setup CSurf

Initalise to set csurf secret in signed httpOnly cookie placed on client machine (place after cookie-parser). The csurf secret allow the server to remain stateless, otherwise the secret will require storing in server session state. The cookie is signed so any tampering will be noticed and exception raised by the cookie-parser:

app.use(csurf({

cookie: {

signed: true,

secure: (app.get('env') === 'production' ? true : false),

httpOnly: true,

sameSite: true

},

value: (req) => {

return req.headers['x-csrf-token']

}

}))

Set CSRF token on page visit by client, which can then be added to headers['x-csrf-token'] and sent AJAX by client:

app.use((req, res, next) => {

res.cookie('XSRF-TOKEN', req.csrfToken(), {

secure: (app.get('env') === 'production' ? true : false),

sameSite: true

})

next()

})

Add error catching, renew XSRF token if failure:

app.use((err, req, res, next) => {

if (err.code !== 'EBADCSRFTOKEN') return next(err)

res.cookie('XSRF-TOKEN', req.csrfToken(), {

secure: true,

sameSite: true

})

debug(err)

res.status(403).send('csrf token invalid')

})

Issue the JWT and XSRF tokens to the client on login

* JWT - SameSite=strict HttpOnly cookies
* XSRF - SameSite cookie, with HMAC signature from server secret

Validate tokens middleware

* Verify JWT token
* Verify Custom Header with XSRF Token

Renew tokens middleware

## Client Side Functions

Transfer XSRF token to HTTP header on all requests (can be limited to restricted requests only, user discretion)

# References:

## Mitigate risk using verfication of Origin

Use server to verify Origin and Referer headers against the target origin (server). If the two match the request is valid since the Origin and Referer headers cannot be manipulated by the browser JavaScript as they are in the forbidden headers list.

<https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name>

If origin header is present make sure it matches target origin. If origin is not present, make sure referer header matches target origin. Matches must be strong for the domain, i.e. example.com does not match example.com.bad.com.

Since application servers often site behind proxies, the request URL cannot always be used to get the target origin. Therefore, it is commonly either:

* set by developer in env
* use the Host header if not changed by the reverse proxy
* use the X-Forwarded-Host header set by reverse proxy if proxy changes host headers

## SameSite Cookie

Most browsers support, however, not all. So should be used along with CSRF tokens to protect users.

## Custom Headers

This defense relies on the same-origin policy (SOP) restriction that only JavaScript can be used to add a custom header, and only within its origin.

<https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html>

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).

## Login/Form Protection

For particular senstive actions reauthentication should be used.

User interaction such as CAPTCHA or reathentication.

Login attacks occur when the attacker tricks a user into sending a signin request for their fake account. Then the attacker can see a users interaction, such as saving a credit card.