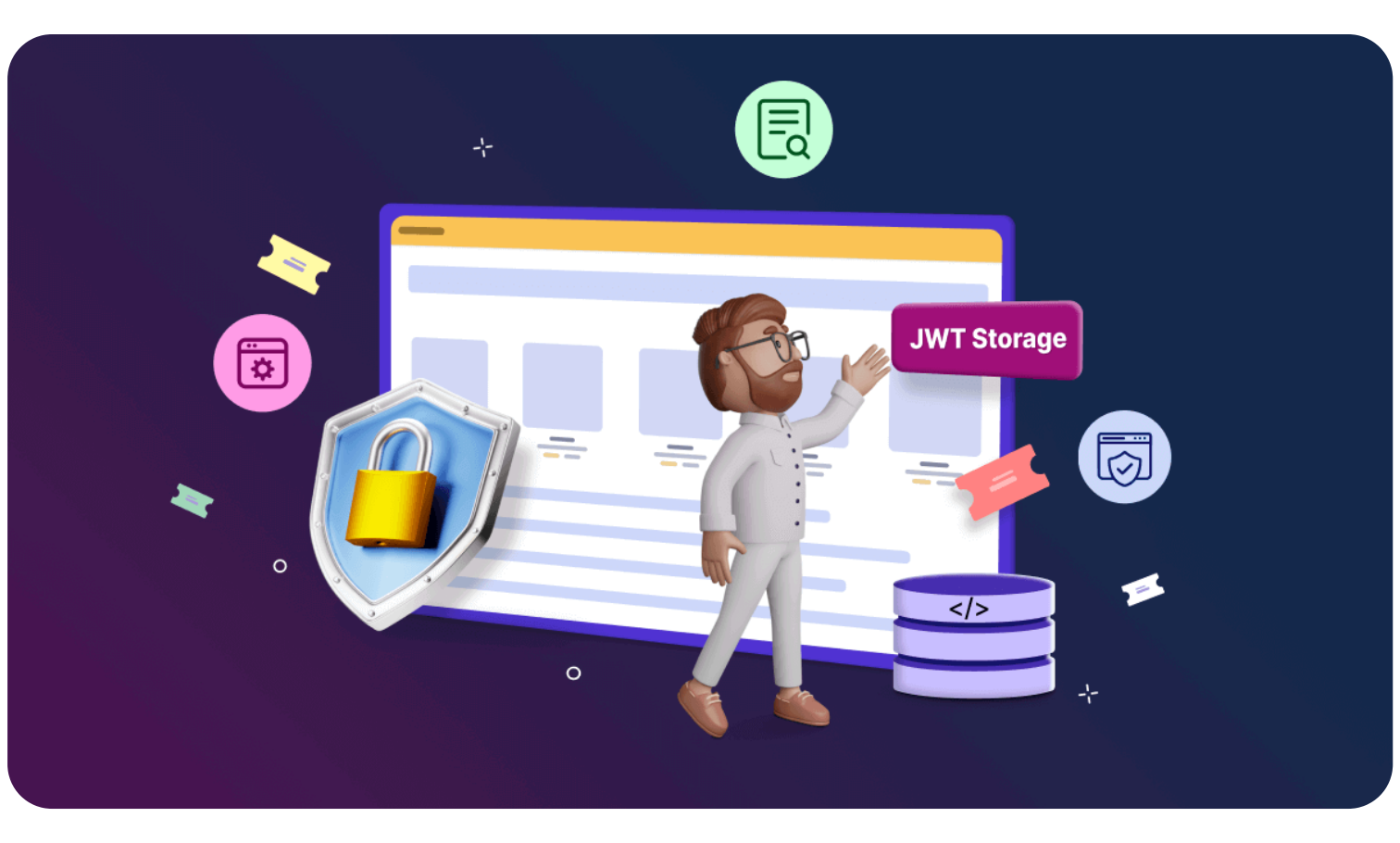
Secure JWT Storage: Best Practices

Binara Prabhanga • 7 min read • Nov 21, 2024



TL;DR: *JSON Web Tokens are widely used for secure data transmission in single-page applications, but they are susceptible to security risks such as cross-site scripting and cross-site request forgery. Secure methods for storing JWTs include using HttpOnly cookies with the Secure flag, encrypting JWTs before storage, and employing server-side session management.*

Single-page applications (SPAs) render only a single HTML page and change the content of that page when the user interacts with it. Angular, React, and Vue.js are some of the most popular SPA frameworks available, and developers often use them for modern web apps, since they enable quick transitions between pages.

However, this wide adaptation has made SPAs a prime target of web attackers, and we need to ensure the security of these apps. We need secure tokens such as JSON Web Tokens (JWT) to do this. A JSON Web Token is a data structure made up of JSON that has been encoded and signed in order to prove that it is genuine and has not been tampered with.

So, let's explore different, secure ways of storing JWT tokens in SPAs, identify potential web threats, and find ways to improve SPA security more.

JWTs and security concerns

JWTs transmit information between clients and servers in a compact JSON format. Despite their convenience, they are susceptible to certain security risks, which can lead to unauthorized access if tokens are mishandled:

- **Cross-site scripting (XSS)**: If JWTs are stored in less secure places, like local storage, attackers can steal them using XSS attacks.
- **Cross-site request forgery (CSRF)**: If the JWT is stored in a cookie that is automatically sent with every request, an attacker could potentially exploit a CSRF vulnerability by tricking the user's browser into making an unauthorized request to the server.

Common storage methods

Local storage and session storage

Local storage is a web browser database that allows you to save data across sessions through key and value pairs. Session storage is like local storage but with a key difference: it stores data only for a browser session. The data is erased once the session ends, typically when the browser tab is closed. Although it is pretty easy to use these storages, they are vulnerable to XSS attacks. If an attacker injects a script, it can access the stored data, which poses a risk when storing sensitive information like JWTs.

Cookies

When configured with security-focused attributes, **HttpOnly**, **Secure**, and **SameSite** cookies offer a more secure alternative. These settings help shield against XSS and CSRF, ensuring cookies are sent over secure connections and are inaccessible to client-side scripts.

Secure methods for storing JWTs

Secure HttpOnly cookies

Storing JWTs in cookies configured with **HttpOnly** and **Secure** flags is a solid approach to enhancing security. This approach improves security by making it impossible for client-side scripts to access the JWTs and only passing them over secure channels:

- **HttpOnly flag:** Setting the **HttpOnlyflag** on cookies means that JavaScript running in the browser can't access them. This is a crucial step in defending against attackers who might try to run malicious scripts to steal your cookies, making it much harder for XSS attacks to succeed.
- **Secure flag:** The **Secure** flag guarantees that the cookie can be sent only through the HTTPS connection. This improves security since you can't send cookies through an unencrypted connection.

Implementation example

Copy

```
Set-Cookie: AuthToken=encryptedJwt; HttpOnly; Secure; Path=/; SameSite=Strict;
```

This cookie configuration ensures that the **AuthToken** is only sent over HTTPS, isn't accessible via JavaScript, and isn't sent along with requests initiated from third-party websites, thanks to the **SameSite=Strict** directive.

Encryption before storage

JSON Web Tokens are encrypted to add a certain level of security even if someone gets ahold of them. They are useless without the decryption key.

Before storing a JWT on the client side, it should be encrypted using a strong encryption algorithm. For example, you can use Bcrypt for **Node.js** apps. The encryption keys should also be stored securely on the server and never exposed to the client or hard-coded in the app.

Refer to the following code example for JWT token encryption.

Copy

```
const CryptoJS = require("crypto-js");
const token = 'your.actual.jwt.token';
const secretKey = 'your.actual.secret-key';

const encryptedToken = CryptoJS.AES.encrypt(token, secretKey).toString();
localStorage.setItem('encryptedToken', encryptedToken);
```

This code example encrypts the JWT with AES and stores the JWT in **localStorage** after creating the encrypted token. Although we use **localStorage** here, no one can use the JWT without the encryption key.

Server-side session management

Another solid approach to storing JWTs is to handle the session only on the server using a unique session ID in a cookie. Every time a new request hits a server, this ID is sent through the cookie to the server, and then the server uses this specific ID to fetch the JWT and check on the session. This approach reduces the risk by keeping the JWT off the client side.

Implementation details

Copy

```
// Server-side
const sessionId = generateSessionId(); // Generate a unique session ID
storeSession(sessionId, jwt); // Store the JWT against the session ID in server storage

// Set cookie with session ID.
res.cookie('sessionID', sessionId, { httpOnly: true, secure: true, sameSite: 'Strict' }

// Client-side
const storedSessionId = document.cookie; // Automatically sent with HTTP requests
```

In this setup, the JWT is securely stored on the server and mapped to a session ID stored in a cookie. The cookie's properties are set up due to the proper transmission and its inability to be read by JavaScript, which adds extra protection.

Advanced security practices

Using web workers for JWT management

Web workers have become an interesting element in web development, while JavaScript can now run in background threads. To enhance security, you can use web workers to implement JWTs. This is effective because it frees a token from the main app environment, which is generally exposed to XSS threats.

Refer to the next example for managing JWTs using web workers.

Copy

```
// In the main application.
var worker = new Worker('jwtWorker.js');

worker.postMessage({jwt: 'your.jwt.token'});

worker.onmessage = function(event) {
  console.log('Received token:', event.data.jwt);
};

// Inside jwtWorker.js
onmessage = function(event) {
  let jwt = event.data.jwt;
  // Perform operations with JWTpostMessage({jwt: jwt});
};
```

This code example shows how a web worker can securely manage JWTs by keeping them out of the global window scope. This reduces the likelihood that extremely dangerous scripts will get a hold of the token, making it secure.

Content security policy (CSP)

You can use CSPs to counter XSS attacks that considerably threaten JWTs. A CSP enables you to specify which resources the browser can load on the page, thus restricting unwanted scripts.

Guidelines for setting a CSP for SPAs

- Use the **default-src** directive to control the default behavior for fetching resources such as JavaScript, images, CSS, and fonts.
- Set **script-src** to restrict where scripts can be loaded, ideally only allowing scripts from your domain and trusted sources.
- Include **object-src 'none'** to block all plugins (like Flash) which can be exploited.

Copy

```
Content-Security-Policy: default-src 'self'; script-src 'self' https://trustedsource.co
```

This CSP header ensures that only scripts from your domain and **trustedsource.com** are allowed, enhancing your SPA's security posture against script injections.

Regular security audits and updates

Maintaining the security of your SPA requires ongoing attention. It is important to conduct regular audits to look for potential gaps and update your libraries and frameworks when needed.

Best practices for security maintenance

- **Schedule regular security audits:** Use automated tools and manual inspections to routinely check your app for potential vulnerabilities.
- **Stay informed on updates:** Get notified on updates for security issues for third-party libraries and frameworks to know when they are out.
- **Implement a quick response process:** Build a procedure that would allow you to change your environment as soon as new security patches come onto the market.

Conclusion

JSON Web Tokens are essential for single-page apps to secure user sessions. However, they come with risks, such as exposure to XSS and CSRF attacks.

So, you need to ensure that you take the necessary security steps, such as proper storage, encryption, server-side session management, web workers, and CSPs, to enhance the security of your SPA.

Well, you don't need to implement all of this. But implement what is necessary based on your app requirements.

Feel free to share your thoughts in the comments section below or reach out to us through our support forum, support portal, or feedback portal. We're always here to help you!

Related blogs

- Secure File Handling in Blazor: Implement JWT Authentication
- Best Practices for JWT Authentication in Angular Apps
- Build CRUD REST APIs with ASP.NET Core 3.1 and Entity Framework Core, Create JWT Tokens, and Secure APIs
- Single Page Application Example using Essential JS 2 Components

Tags:

Development | JSON | JWT | Security | Single-Page Applications | Web

MEET THE AUTHOR

Binara Prabhanga

Experienced software developer proficient in JavaScript, C, and Java with a passion for writing. Tech writer since 2021, creating engaging and informative content for a variety of audiences.

Leave a comment

You must be Logged in to post a comment.

Submit Comment