LocalStorage vs Cookies: the best-practice guide to storing JWT tokens securely in your front-end

By Nick Faro

Last Updated On: Monday, September 2, 2024

In today's web development landscape, JSON Web Tokens (JWT) have become a popular choice for authentication and authorization. However, securely storing JSON web tokens in an application's frontend poses a significant challenge.

In this article, we will explore various techniques to address this issue and ensure the protection of sensitive user information.

We will cover the pros and cons of using `localStorage` and cookies and provide code snippets to implement these solutions effectively.

What is a JSON web token (JWT token)?

Before delving into storage options, it's crucial to understand the nature of a JWT token – its an Open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.[1]

A JSON web token consists of three parts: a header (think authorization header), a JWT payload, and a signature.

The payload contains claims, such as user information or permissions (eg. used as an access token), while the signature ensures the token's integrity.

What are the security concerns with storing JSON web tokens in an application frontend?

When it comes to storing JSON web tokens in the frontend, two primary concerns arise:

1. Protection against XSS attacks (cross-site scripting) and
2. Mitigation of token theft.

Cross-Site Scripting (XSS) attacks with insecure JWT token storage

XSS attacks occur when an attacker injects malicious code into a website, gaining unauthorized access to sensitive data.

Storing these tokens in insecure locations can make them vulnerable to XSS.

JWT token theft

If an attacker manages to obtain a user's JWT token, they can impersonate the user and gain unauthorized access to protected resources.

Therefore, it is essential to employ secure storage techniques to prevent token theft.

Storing a JSON web token in Local Storage

`localStorage` is a built-in browser storage mechanism that allows web applications to store data persistently. However, it is crucial to consider its advantages and disadvantages when using it to store a JSON web token.

Pros of LocalStorage

- Simplicity: The `localStorage` API is straightforward to use, making implementation easier.
- Persistence: Data stored in `localStorage` remains available even after the user closes the browser or reboots the system.

Cons of LocalStorage

- XSS attack: storing JSON web tokens in `localStorage` makes them susceptible to a XSS attack.
- Lack of Encryption: LocalStorage does not provide built-in encryption, encrypted tokens make the stored data virtually inaccessible if an attacker gains access to the user's device.

Storing JSON web tokens in cookies

Cookies are another popular storage mechanism for web applications. You get certain advantages when you use cookies for storing JSON web tokens:

Pros of cookies

- Built-in Security: Cookies provide a built-in `secure` flag that allows only encrypted transmission over HTTPS.
- Same-Origin Policy: Cookies are subject to the `same-origin` policy, which helps mitigate XSS vulnerabilities.
- Options for `Secure` and `HttpOnly` Flags: By setting the secure flag and HttpOnly flag on cookies, you can enhance their security.

Cons of cookies

- Complexity: working with cookies can be more complex compared to LocalStorage.
- Limited Storage: cookies have a size limit of approximately 4KB, which may pose a constraint when storing large JWT tokens.

Best Practices for securely storing JSON web tokens

To securely store a JSON web token in the frontend, consider the following best practices:

1. Encryption: If you choose to use LocalStorage, encrypt the JWT tokens before storing them to enhance their security. Various encryption libraries and algorithms are available for this purpose.
2. Short validity: Set a short lifespan for JWT tokens to minimize the window of opportunity for attackers to exploit stolen tokens.
3. Refresh token: a mechanism that utilizes simple web tokens to refresh tokens and reject tokens that have expired will help to protect your user's data and minimize the chances of data theft.
4. `Secure` and `HttpOnly` flags: if you opt for cookies, utilize the secure and `HttpOnly` flags to enhance their security. The secure setting in a cookie ensures transmission only over HTTPS, while the `HttpOnly` flag forbids JavaScript code from accessing the cookie, for example, through the `Document.cookie` property.

Preferred Approach: Storing JSON web tokens with cookies for persistent storage

Cookies provide enhanced security, compatibility, session persistence, and scalability, making them the preferred option for persistently storing JSON web tokens in the frontend of a Nodejs web application for the following reasons:

Enhanced security features

Cookies offer built-in security features such as the `secure` setting for encrypted transmission over HTTPS and the `HttpOnly` flag to prevent client-side JavaScript code access, minimizing the risk of token theft through XSS attacks.

Here's an example of how to set these flags in a Node.js application using the cookie package:

```
const cookie = require('cookie');

// Set a cookie with the Secure and HttpOnly flags
const cookieOptions = {
  secure: true, // Secure
  httpOnly: true, // HttpOnly
  maxAge: 60 * 60 * 24 * 7, // 1 week
  sameSite: 'strict',
  path: '/', // Cookie is available for all paths
};

const cookieString = cookie.serialize('jwtToken', token, cookieOptions);

// Set the cookie in the response header
res.setHeader('Set-Cookie', cookieString);
```

Same-Origin policy enforcement

Cookies adhere to the same-origin policy, limiting their access to the originating domain. This strengthens protection against a XSS attack and makes it harder for attackers to compromise tokens.

Support for token expiration and revocation

Cookies support setting expiration dates, enforcing token validity periods, and can be easily invalidated on the server side to revoke access if necessary. Here's an example of setting an expiration date for a cookie:

```
const cookie = require('cookie');

// Set the expiration date to 1 hour from now
const expirationDate = new Date();
expirationDate.setTime(expirationDate.getTime() + 1 * 60 * 60 * 1000);

const cookieOptions = {
  expires: expirationDate,
  // ...
};

const cookieString = cookie.serialize('jwtToken', token, cookieOptions);

// Set the cookie in the response header
res.setHeader('Set-Cookie', cookieString);
```

To revoke a cookie, you can set its expiration date to a past date, rendering it invalid.

Compatibility with cross-domain requests

Cookies can be sent with requests to different domains, facilitating authentication and authorization in cross-domain scenarios. This behavior is achieved by configuring the CORS (Cross-Origin Resource Sharing) settings on the server.

Here's an example using the CORS package in a Node.js application:

```
const cors = require('cors');

// Enable CORS for all routes and allow credentials cookies
app.use(cors({ origin: true, credentials: true }));
```

Make sure to replace `http://example.com` with the appropriate domain or origins that should be allowed to make cross-domain requests.

Persistence across browser sessions

Unlike `localStorage`, cookies persist across browser sessions, ensuring users remain authenticated even after closing and reopening the browser.

Scalability for large tokens

Cookies have a larger storage capacity compared to `LocalStorage`, making them suitable for storing larger JSON web tokens or additional metadata.

Optimal Secure Solution: Save JWT Tokens in the browser's memory and store the refresh token in a cookie

When it comes to securely storing this type of access token in your web application, an optimal solution is to save the token in browser session storage while storing the refresh token in a cookie protected by the `secure` and `HttpOnly` settings.

This approach offers a balance between security and convenience. The JSON web token in session storage provides quick access during the user's session, while the refresh token in a cookie ensures long-term persistence and protection against CSRF attacks.

How does this approach help mitigate a CSRF attack?

Even if a new refresh token is generated by the attacker, they won't be able to read the response if they're using an HTML form.

It's important to understand that preventing attackers from making a succesful fetch or AJAX request in order to read the response requires that your authorization server's CORS policy is set up correctly to prevent requests from unauthorized websites.

Here's a step-by-step guide with Node.js code snippets for each step:

Step 1: Generate and issue tokens

When a user successfully authenticates, generate both a JSON web token and a refresh token on the server-side. The JSON token contains short-lived access information, while the refresh token is a long-lived token used for obtaining new JSON web tokens when they expire.

```
// Generate JWT token
const jwt = require('jsonwebtoken');

const payload = { userId: 'user123' };
const secretKey = 'your-secret-key';
const jwtToken = jwt.sign(payload, secretKey, { expiresIn: '1h' });

// Generate refresh token
const refreshToken = jwt.sign(payload, secretKey, { expiresIn: '7d' });

// Return both tokens to the client
res.json({ jwtToken, refreshToken });
```

Step 2: Save the JSON web token in the browser session

On the client-side, save the JSON web token in browser session storage upon successful authentication. This ensures the token remains available during the user's session but is cleared when the browser tab is closed.

```
// Save JWT token in session storage
sessionStorage.setItem('jwtToken', jwtToken);
```

Step 3: Save the refresh token in a secure HttpOnly Cookie

This ensures it is securely stored and inaccessible to client-side JavaScript code. It also helps protect against a CSRF attack.

This step utilises 3 key cookie-specific security controls:

- The `httpOnly` flag to prevent JavaScript from reading it.
- The `secure=true` flag so it can only be sent over HTTPS.
- The `SameSite=strict` flag whenever possible to prevent CSRF. This can only be used if your Authorization Server has the same domain as your front end. If the domains are different, your Authorization Server must set CORS headers in the backend or use other methods to ensure that the refresh token request can only be successfully performed by authorized websites.

```
// Set refresh token as an HttpOnly cookie
res.cookie('refreshToken', refreshToken, {
  secure: true, // Set to true if using HTTPS
  httpOnly: true,
  sameSite: 'strict', // Adjust to your requirements
  maxAge: 7 * 24 * 60 * 60 * 1000, // Set the expiration time (7 days in this example)
});
```

Step 4: How to refresh the JSON web tokens

When the JSON web token expires, the client can use the refresh token stored in the cookie to request a new JSON web token from the server.

This process ensures continuous authentication without requiring the user to manually log in again.

```
// Refreshing JWT token using the refresh token
const cookie = req.cookies.refreshToken;

if (cookie) {
  jwt.verify(cookie, secretKey, (err, decoded) => {
    if (err) {
      // Handle invalid or expired refresh token
      res.status(401).json({ error: 'Invalid or expired refresh token' });
    } else {
      // Generate a new JWT token
      const newJwtToken = jwt.sign({ userId: decoded.userId }, secretKey, { expiresIn: '1h' });

      // Update the JWT token in session storage
      sessionStorage.setItem('jwtToken', newJwtToken);

      // Return the new JWT token to the client
      res.json({ jwtToken: newJwtToken });
    }
  });
} else {
  // No refresh token found, prompt user to log in
  res.status(401).json({ error: 'Refresh token not found' });
}
```

How to test security of tokens and cookies?

There are very few tools that allow software developers to protect their web applications against hackers - without the help of specialist application security experts.

However, Cyber Chief is one such tool that allows you to run regular vulnerability scans with a web app vulnerability scanning tool.

See how Cyber Chief works now to see not only how it can help to keep attackers out, but also to see how you can ensure that you ship every release with zero known critical vulnerabilities like CSV formula injection attacks and thousands more.

Cyber Chief helps you run automated tests for your web app security, API security and cloud infrastructure security and each subscription comes with:

- Results from scanning your application for the presence of OWASP Top 10 + SANS CWE 25 + thousands of other vulnerabilities.
- A detailed description of the vulnerabilities found.
- A risk level for each vulnerability, so you know which GraphQL endpoints to fix first.
- Best-practice fixes for each vulnerability, including code snippets where relevant.
- On-demand security coaching support from our application security experts to help you patch vulnerabilities faster.

Here's what you can do next.

See Cyber Chief In Action >> | **Run Express Scan On Your App**

Contacts

Australia:
+61 3 7001 1430
Singapore:
+65 3159 4054
United Kingdom:
+44 20 3411 4974
United States:
+1 650 278 4438
International:
+61 3 7001 1430
Email:
solutions[at]audacix.com
.
Cyber Chief Is A Product Of
Audacix