

Analysis of OAuth 2.0 Vulnerabilities Arising from Weak Implementation Choices

Leo Petrović¹

¹Faculty of Mechanical Engineering, Computing and Electrical Engineering University of Mostar, Bosnia and Herzegovina

Abstract This project showcases authentication and authorization frameworks, such as OAuth 2.0 and OpenID Connect, by implementing a simplified OAuth 2.0 system. To illustrate possible attacks on such a system, a demonstration project is implemented using an incorrectly configured OAuth 2.0 authentication flow and an insecure OAuth client. Solutions are presented that both prevent potential attacks and protect user data, even in the event of a successful attack. The demonstration shows that a maliciously injected script can read a user's access token and send it to the attacker, who can use it to access the user's private data, effectively hijacking the session. This setup demonstrates that, while OAuth 2.0 provides a secure protocol, security is undermined by weak implementation choices. In particular, storing tokens in localStorage and allowing XSS in the client can completely defeat OAuth's protections. The findings emphasize that protocol security does not guarantee overall system security without secure practices.	Article history Received: 9. 9. 2025. Revised: 3.11. 2025. Accepted: 7. 11. 2025. Keywords Authentication, OAuth 2.0, Access Token Theft, Cross-Site Scripting, React, LocalStorage.
--	--

1 Introduction

OAuth 2.0 is a widely adopted framework that enables secure user authorization across web and mobile applications. Instead of sharing credentials directly, users can grant third-party applications limited access to their data on a single platform through an intermediary authorization process. This approach underpins the "Log in with Google" or "Sign in with Facebook" features common across the web [1], [2]. The protocol's design is generally regarded as secure; however, its flexibility and reliance on implementation details often introduce critical security risks. As PortSwigger observes, OAuth 2.0's adaptable structure makes it "inherently prone to implementation mistakes", and these mistakes can expose users to severe vulnerabilities [2], [3].

Over the past decade, researchers and security analysts have demonstrated that improperly configured OAuth flows can allow attackers to intercept tokens, impersonate users, or access confidential data [2], [3], [4].

The variety of available OAuth "flows" (or grants) contributes to this complexity. Each flow defines a distinct method for obtaining an access token, and choosing the correct one depends on factors such as the application type, the level of trust in the client, and the desired user experience [4]. For example, financial institutions often employ a hardened version of OAuth known as the Financial-grade API (FAPI) to meet stringent security and compliance requirements [5].

In this study, we focus primarily on the Authorization Code flow, which serves as the basis for many real-world implementations. While this flow can be secure when paired with Proof Key for Code Exchange (PKCE), omitting PKCE introduces vulnerabilities that make it valuable for demonstrating common misconfigurations [4]. By examining this flow in a controlled environment, we show how subtle implementation errors – such as insecure token storage or improper handling of user input – can compromise even the most well-designed authentication protocols.

Contact Leo Petrović, leo.petrovic@fsre.sum.ba, Faculty of Mechanical Engineering, Computing and Electrical Engineering, University of Mostar

©2026 by the Author(s). Licensee IJISE by Faculty of Mechanical Engineering, Computing and Electrical Engineering, University of Mostar. This article is an open-access and distributed under the terms and conditions of the CC BY 4.0 (<https://creativecommons.org/licenses/by/4.0/>)

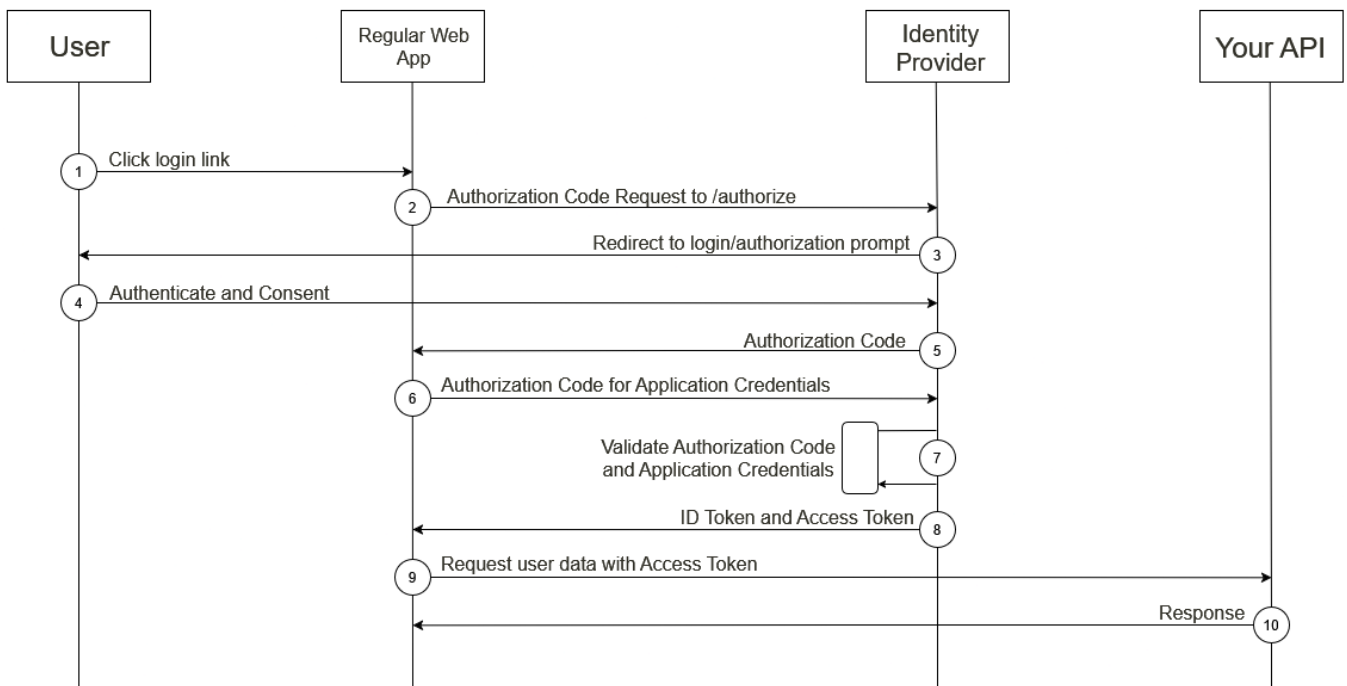


Figure 1 OAuth 2.0 Authorization Code Flow

1.1 OAuth 2.0 Flows

The OAuth 2.0 Authorization Framework supports several different "flows" (or grants). Flows are ways of retrieving an access token. Deciding which one is suited for any use case depends mainly on the application type, but other parameters weigh in as well, like the level of trust for the client, or the experience the users have [4]. In banking systems, for example, a completely different authentication approach based on OAuth 2.0 (called FAPI) is used to harden the system [6].

This project primarily focuses on the OAuth 2.0 Authorization Code flow (without PKCE), as it is simple, which can serve well in the demonstration, though this flow is insecure in practice if not used with PKCE [4]. The basic principle of how this flow works is shown in Figure 1 - OAuth 2.0 Authorization Code Flow.

1.2 Security Context

One common pitfall in OAuth deployments is insecure token storage on the client side. Web single-page applications (SPAs) typically cannot store secrets, so developers sometimes keep tokens in places like localStorage for simplicity [7]. OWASP warns that public clients "do not have the possibility of storing

tokens securely" because scripts (e.g., via XSS) could access these tokens [5][8]. Indeed, if an application renders user-supplied HTML (for example, using React's dangerouslySetInnerHTML) without sanitization, an attacker can inject code that reads and exfiltrates tokens [5], [9], [10]. Such cross-site scripting (XSS) vulnerabilities are ranked among the most critical web risks and can completely undermine OAuth security [11].

2 Methodology

2.1 Introduction

In this project, a simplified OAuth 2.0 system was constructed to demonstrate these risks. We implement an authorization server with only the /authorize, /token, and /register endpoints (omitting PKCE and refresh tokens), and an API that returns protected user data when provided a valid access token. A React client runs a login flow and stores the access token in localStorage, and includes a "public note" feature that dangerously renders unsanitized HTML. We also include an "evil" server that collects stolen tokens and uses them to call the API as the victim, demonstrating how a malicious note can steal a token and allow the attacker to access the victim user's protected resources [2], [11]. The

project directory structure is shown in Figure 2: Relevant Project Directory Structure. Ultimately, this demonstration underscores that OAuth's protocol security can be negated by poor frontend design: insecure token storage and XSS make the system completely vulnerable [7], [10], [12], [13].

2.2 Attack Setup and Injection

In our simulated attack, the single-page client application includes a "public note" feature that renders user-supplied HTML via React's `dangerouslySetInnerHTML` without sanitization. This is inherently dangerous: as one security analysis notes, using `dangerouslySetInnerHTML` on untrusted input is "insanely insecure" [9]. Because it will execute any embedded HTML or scripts on the page. In this scenario, an attacker crafts a malicious HTML payload containing a `<script>` tag. The attacker then posts this payload (or sends it via a link) so that the victim's browser will load the note. Because the application blindly injects the note's content into the DOM, the attacker's script is injected into the origin of the client application.

2.3 Script Execution and Token Theft

When the victim user views the malicious note, the embedded script runs immediately in the context of the client application. Crucially, the client had previously obtained an OAuth 2.0 access token (for example, during login) and stored it in `window.localStorage` to use for API calls. Any script running on the page (whether legitimate or injected) can read `localStorage`.

Thus, as soon as the malicious script executes, it reads the access token from `localStorage` (e.g., via `window.localStorage.getItem("token")`). This kind of XSS-based theft of tokens is a known risk: if an attacker can inject JavaScript into a page, they can steal any access token in `localStorage` [5], [8], [14].

The OWASP testing guide specifically warns that public clients using `localStorage` are vulnerable to XSS, noting that "a cross-site scripting attack allows attackers to access credentials stored in the browser" [12]. In our simulation, the injected script thus acquires the victim's access token.

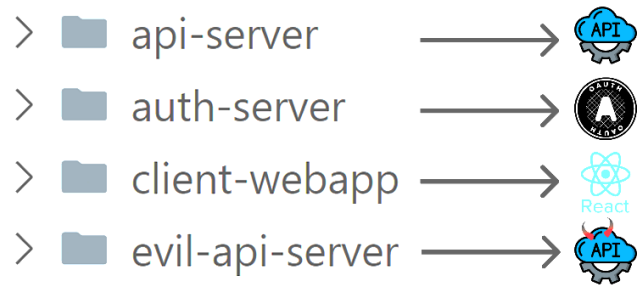


Figure 2 Relevant project directory structure

2.4 Token Exfiltration and API Abuse

Once the malicious script has extracted the victim's token, it exfiltrates it to the attacker's server. For example, the script might create an HTTP GET or POST request, as shown in Figure 3.

This causes the user's browser to send the token in a request to the attacker's server, which logs it. With the token in hand, the attacker can now impersonate the victim to the resource server. The attacker's server uses the stolen token as a Bearer credential in an API request (e.g., `Authorization: Bearer <token>`) to the protected resource endpoint. Since the token is valid and unexpired, the API server accepts it and returns the victim's protected data. In effect, the attacker "becomes" the user: OAuth 2.0 tokens are bearer tokens, so "those who hold the token can use it" to access resources [15]. This chain of events – from the injection of malicious HTML to XSS execution, token reading, exfiltration, and token misuse – completes the attack. In our demo, we walk through each of these steps in code, illustrating how a single XSS flaw breaks the security of the OAuth flow.

```
<script>
  // Example of exfiltration
  // (in practice, hidden in note HTML)
  fetch(
    "http://evil.example.com/log?token="
    +
    encodeURIComponent(
      window.localStorage.getItem("token")
    )
  );
</script>
```

Figure 3 Example code that sends a token to the server

3 Demonstration

3.1 OAuth 2.0 Authorization Server

The demo implements an OAuth 2.0 **authorization server** using the Bun JavaScript runtime and the Hono web framework. (Bun is a new JS runtime similar to Node.js [16]. Hono is a lightweight framework that runs on Bun.) The auth server exposes the standard endpoints: GET /authorize (to initiate authorization), POST /token (to exchange an authorization code for tokens), and POST /register (to dynamically register a new client). We use the Authorization Code flow without PKCE for demonstration purposes. This wouldn't usually be done since our client is a browser-based SPA that cannot keep a secret [4], [17]. In practice, this means the client sends a code_challenge in the /authorize request and later proves it by sending a code_verifier with the /token request. (The server checks the verifier's hash to bind the code to the client.) Either way, upon successful token issuance, the /token endpoint returns a response such as the one shown in Figure 4.

The client then uses this access token (a signed JWT in our implementation) to call protected APIs. (For brevity, the server in our demo does not issue refresh tokens, focusing instead on access tokens.)

3.2 Resource API Server

A separate Bun/Hono service acts as the **resource (API) server**. This server's endpoints check incoming requests for a valid access token (for example, in the Authorization: Bearer <token> header). If the token is present and valid, the API returns protected user data (e.g., the user's profile JSON). An example of a GET request to /api/userinfo with a valid token is shown in Figure 5.

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "access_token": "ABC123",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

Figure 4 Example HTTP response from authorization server

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "user": "alice",
  "email": "alice@example.com"
}
```

Figure 5 Example HTTP response from API server

If the token is missing or invalid, the API returns an error (401 Unauthorized). In the attack scenario, once the attacker's server has stolen the victim's token, it will send such an API request using that token to retrieve the user's data.

3.3 Client Application (React + Vite)

The **client** is a React single-page application built with Vite. It has a "Login" button that starts the OAuth flow; clicking it redirects the browser to a specially crafted URI on the authorization server. After the user logs in, the auth server redirects back to the client's callback route with a code. The client's callback handler then sends a POST to /token (as above) to obtain the access_token. The returned token is stored in the window.localStorage, allowing it to be used in subsequent API calls. (Note: storing in localStorage is easy but insecure; any script on the page could read it [18].)

The app also includes the vulnerable "public note" feature. Users can submit a text note (e.g., via an HTML form), and the app renders it on the page. Crucially, the rendering code uses dangerouslySetInnerHTML without sanitizing the input. This means if the input contains any <script> tags or onerror attributes, they will execute in the React app. As noted earlier, this use of dangerouslySetInnerHTML on untrusted content is extremely dangerous [9]. In our demo, the attacker exploits this by inserting malicious HTML into a note. When any user views that note, the embedded JavaScript runs and reads the stored access token.

3.4 Malicious ("Evil") Server

The "evil" server is a simple web service (written in Bun) that simulates the attacker's backend. It exposes an endpoint, such as GET /log, that accepts a query parameter (e.g.,?token=...). When a request arrives, it logs the token and then immediately uses it to call the protected API.

```
// Pseudo-code for the malicious server
app.get('/log', (req, res) => {
  const token = req.query.token;
  console.log("Stolen token:", token);
  // Use token to call API as victim:
  fetch('https://api.example.com/userinfo', {
    headers: { 'Authorization': 'Bearer ' +
token }
  })
  .then(apiRes => apiRes.json())
  .then(data => console.log("API response:",
data));
  res.send("OK");
});
```

Figure 6 Example code for using a stolen access token

A pseudocode example of how this can be achieved is shown in Figure 6, which illustrates the code for using a stolen access token.

This shows that once the token is received, the attacker can use it in an Authorization header to retrieve the user's data from our API (as though they were the legitimate client) [2]. What the attacker sees is shown in Figure 7.

4 Mitigation

To mitigate such risks, several best practices should be followed. First, avoid storing tokens in client-side scripts-accessible locations. The OWASP community specifically recommends using secure, HttpOnly cookies for storing session tokens instead of localStorage [12], [14]. Cookies with the HttpOnly flag cannot be read by JavaScript, preventing XSS from stealing them. Second, deploy a strict Content

```
Started server http://localhost:5002
User access token:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiJlNTI4MjMNS05MGRmLTRmYzItOTkzOC1hYjRi
OTE2YmU
xZjkiLCJhdWQiOiJteS1jbGllbnQtaWQiLCJzY29wZSI6InJ
lYWQiLC
JleHAiOiJlNTI4MjMNS05MGRmLTRmYzItOTkzOC1hYjRi
9KRl3IB
048KXR7y3BkwtE7iZoQIKrES5mBGpFKCkI
User data: {
  sub: "e52822f5-90df-4fc2-9938-ab4b916be1f9",
  email: "leopetrovic11@gmail.com",
}
Private data: Hello user with ID e52822f5-90df-
4fc2-9938
-ab4b916be1f9
```

Figure 7 Results of a successful attack

Security Policy (CSP) that disables inline scripts and limits script sources [4], [5]. A well-configured CSP can prevent injected scripts from running even if they are inserted into the DOM [18], [19]. Third, always sanitize or avoid rendering raw HTML. Instead of using dangerouslySetInnerHTML, user input should be passed through a robust sanitizer (for example, DOMPurify) before insertion [9], or HTML rendering should be avoided altogether in favor of safer approaches. Shorter token lifetimes and the requirement of one-time use of long-lived tokens should also be enforced [4]. As noted by security guides, access tokens should have short expirations (hours or days) so that a stolen token quickly becomes useless [15]. Refresh tokens (if used) should be rotated and limited to mitigate theft. Finally, the most secure way to implement OAuth 2.0 is to use an existing, battle-tested implementation. Around 10% of all websites using OAuth 2.0 are vulnerable simply because they deviate from the standard protocol [20].

5 Conclusion

This demonstration highlights how client-side implementation flaws can undermine a seemingly secure OAuth 2.0 setup. We showed that an XSS vulnerability in a SPA – specifically, unsanitized HTML rendering – can allow an attacker to steal an OAuth access token from localStorage and use it to access protected resources as the victim. Notably, this does not violate the OAuth protocol itself; instead, it exploits the token after it has been issued. The lesson is that OAuth's security guarantees assume tokens remain confidential. If tokens are stored where injected scripts can access them, an attacker only needs one flaw (in this case, dangerouslySetInnerHTML) to break the system.

In summary, the protocol-level measures in OAuth 2.0 do not automatically protect against implementation flaws. Our proof-of-concept deliberately included insecure practices (like storing tokens in localStorage and rendering unsanitized HTML) to make the attack clear. This demo is not production-ready; it is a teaching tool. In a real application, using secure storage (HttpOnly cookies), strict CSP, input sanitization, and other defenses is essential to prevent XSS-based token theft. By combining these

countermeasures, developers can uphold OAuth 2.0's promise of secure authorization in practice.

Conflicts of Interest: The author reports there are no competing interests to declare;

6 References

- [1] OWASP Foundation, "OAuth 2.0 Security Cheat Sheet," OWASP Cheat Sheet Series.
- [2] W. Li and C. J. Mitchell, "Security Issues in OAuth 2.0 SSO Implementations," 2014, pp. 529–541. doi: 10.1007/978-3-319-13257-0_34.
- [3] L. Weichselbaum, "Mitigate cross-site scripting (XSS) with a strict Content Security Policy (CSP)," Google Web.Dev Blog.
- [4] Ed. D. T. A. T. Lodderstedt, I. M. McGloin, and O. C. P. Hunt, *OAuth 2.0 Threat Model and Security Considerations*. IETF Trust, 2013.
- [5] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffei, "Discovering concrete attacks on website authorization by formal analysis," *J Comput Secur*, vol. 22, no. 4, pp. 601–657, Apr. 2014, doi: 10.3233/JCS-140503.
- [6] D. Fett, P. Hosseini, and R. Kuesters, "An Extensive Formal Security Analysis of the OpenID Financial-grade API," 2019. [Online]. Available: <https://arxiv.org/abs/1901.11520>
- [7] S.-T. Sun and K. Beznosov, "The devil is in the (implementation) details," in *Proceedings of the 2012 ACM conference on Computer and communications security*, New York, NY, USA: ACM, Oct. 2012, pp. 378–390. doi: 10.1145/2382196.2382238.
- [8] OWASP Foundation, "Cross Site Scripting (XSS) – OWASP," OWASP.org.
- [9] OWASP Foundation, "Content Security Policy Cheat Sheet," OWASP Cheat Sheet Series.
- [10] J. Catalan and S. Drosdzol, "Common OAuth Vulnerabilities," Doyensec.
- [11] E. Ferry, J. O Raw, and K. Curran, "Security evaluation of the OAuth 2.0 framework," *Information & Computer Security*, vol. 23, no. 1, pp. 73–101, Mar. 2015, doi: 10.1108/ICS-12-2013-0089.
- [12] P. Philippaerts, D. Preuveneers, and W. Joosen, "OAuch: Exploring Security Compliance in the OAuth 2.0 Ecosystem," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, New York, NY, USA: ACM, Oct. 2022, pp. 460–481. doi: 10.1145/3545948.3545955.
- [13] W. Li, C. J. Mitchell, and T. Chen, "Your Code Is My Code: Exploiting a Common Weakness in OAuth 2.0 Implementations," in *Security Protocols XXVI: 26th International Workshop, Cambridge, UK, March 19–21, 2018, Revised Selected Papers (pp.24–41)*, 2018, pp. 24–41. doi: 10.1007/978-3-030-03251-7_3.
- [14] D. Fett, R. Kuesters, and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0," Aug. 2016.
- [15] E. Shernan, H. Carter, D. Tian, P. Traynor, and K. Butler, "More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations," 2015, pp. 239–260. doi: 10.1007/978-3-319-20550-2_13.
- [16] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA: ACM, Oct. 2016, pp. 1376–1387. doi: 10.1145/2976749.2978363.
- [17] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th international conference on World wide web*, New York, NY, USA: ACM, Apr. 2010, pp. 921–930. doi: 10.1145/1772690.1772784.
- [18] G. Fors and A. Radhi, "Security and performance impact of client-side token storage methods," 2022. Accessed: Sep. 12, 2025. [Online]. Available: <https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1676749&dswid=-3883>
- [19] A. Hannousse, S. Yahiouche, and M. C. Nait-Hamoud, "Twenty-two years since revealing cross-site scripting attacks: a systematic mapping and a comprehensive survey," May 2022, doi: 10.1016/j.cosrev.2024.100634.
- [20] S. Calzavara, R. Focardi, M. Maffei, C. Schneidewind, M. Squarcina, and M. Tempesta, "WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 1493–1510. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity18/presentation/calzavara>