
Stream: Internet Engineering Task Force (IETF)
RFC: [10017](#)
BCP: 212
Category: Best Current Practice
Published: August 2026
ISSN: 2070-1721
Authors: A. Parecki P. De Ryck D. Waite
Okta Pragmatic Web Security Ping Identity

RFC 10017

OAuth 2.0 for Browser-Based Applications

Abstract

This specification details the threats, attack consequences, security considerations, and best practices that must be taken into account when developing browser-based applications that use OAuth 2.0.

Status of This Memo

This memo documents an Internet Best Current Practice.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on BCPs is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at <https://www.rfc-editor.org/info/rfc10017>.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	4
2. Notational Conventions	5
3. Terminology	5
4. History of OAuth 2.0 in Browser-Based Applications	6
5. The Threat of Malicious JavaScript	6
5.1. Attack Scenarios	8
5.1.1. Single-Execution Token Theft	8
5.1.2. Persistent Token Theft	8
5.1.3. Acquisition and Extraction of New Tokens	9
5.1.4. Proxying Requests via the User's Browser	10
5.2. Attack Consequences	10
5.2.1. Exploiting Stolen Refresh Tokens	11
5.2.2. Exploiting Stolen Access Tokens	11
5.2.3. Client Hijacking	11
6. Application Architecture Patterns	12
6.1. Backend for Frontend (BFF)	12
6.1.1. Application Architecture	13
6.1.2. Implementation Details	14
6.1.3. Security Considerations	16
6.1.4. Threat Analysis	20
6.2. Token-Mediating Backend	22
6.2.1. Application Architecture	22
6.2.2. Implementation Details	24
6.2.3. Security Considerations	25
6.2.4. Threat Analysis	26
6.3. Browser-Based OAuth 2.0 Client	28
6.3.1. Application Architecture	28
6.3.2. Implementation Details	29

6.3.3. Security Considerations	31
6.3.4. Threat Analysis	32
7. Discouraged and Deprecated Architecture Patterns	35
7.1. Single-Domain Browser-Based Applications (Not Using OAuth)	35
7.1.1. Threat Analysis	36
7.2. OAuth Implicit Grant	36
7.2.1. Historic Note	36
7.2.2. Threat Analysis	36
7.2.3. Further Attacks on the Implicit Grant	37
7.2.4. Disadvantages of the Implicit Grant	38
7.3. Resource Owner Password Credentials Grant	38
7.4. Handling the OAuth Flow in a Service Worker	38
7.4.1. Threat Analysis	39
8. Token Storage in the Browser	41
8.1. Cookies	41
8.2. Token Storage in a Service Worker	42
8.3. Token Storage in a Web Worker	42
8.4. In-Memory Token Storage	43
8.5. Persistent Token Storage	43
8.6. Filesystem Considerations for Browser Storage APIs	44
9. Security Considerations	44
9.1. Reducing the Authority of Tokens	44
9.2. Sender-Constrained Tokens	44
9.3. Authorization Server Mix-Up Mitigation	45
9.4. Isolating Applications Using Origins	45
10. IANA Considerations	45
11. References	46
11.1. Normative References	46
11.2. Informative References	47
Acknowledgements	48

1. Introduction

This specification describes different architectural patterns for implementing OAuth 2.0 clients in applications executing in a browser. It also outlines the security challenges for browser-based applications and analyzes how different patterns can help address some of these challenges.

This document focuses on JavaScript frontend applications acting as the OAuth client (defined in [Section 1.1](#) of [\[RFC6749\]](#)), interacting with the authorization server ([Section 1.1](#) of [\[RFC6749\]](#)) to obtain access tokens and optionally refresh tokens. The client uses the access token to access protected resources on resource servers ([Section 1.1](#) of [\[RFC6749\]](#)). When using OAuth, the client, authorization server, and resource servers are all considered independent parties, regardless of whether each is owned or operated by the same entity.

Note that many web applications consist of a frontend and API running on a common domain, allowing for an architecture that does not rely on OAuth 2.0. This is described in more detail in [Section 7.1](#). Such scenarios can rely on OpenID Connect [\[OpenID\]](#) for federated user authentication, after which the application maintains the user's authentication state. Such a scenario (which only uses OAuth 2.0 as the underlying specification of OpenID Connect) is not within scope of this specification.

For native application developers using OAuth 2.0 and OpenID Connect, an IETF BCP (Best Current Practice) was published that guides integration of these technologies. That document is formally known as [\[RFC8252\]](#) or BCP 212 but is often referred to as "AppAuth" after the OpenID Foundation-sponsored set of libraries that assist developers in adopting these practices. [\[RFC8252\]](#) makes specific recommendations for how to securely implement OAuth clients in native applications, including incorporating additional OAuth extensions where needed.

This specification, "OAuth 2.0 for Browser-Based Applications", highlights how the security properties of browser-based applications are vastly different than those of native applications, and it addresses the similarities between implementing OAuth clients as native applications and browser-based applications. This document is primarily focused on OAuth, except where OpenID Connect provides additional considerations.

Many of these recommendations are derived from the Best Current Practice for OAuth 2.0 Security [\[RFC9700\]](#), as browser-based applications are expected to follow those recommendations as well. This document expands on and further restricts various recommendations given in [\[RFC9700\]](#).

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [[RFC2119](#)] [[RFC8174](#)] when, and only when, they appear in all capitals, as shown here.

3. Terminology

This specification uses the terms "access token", "authorization endpoint", "authorization grant", "authorization server", "client", "client identifier" (client ID), "protected resource", "refresh token", "resource server", and "token endpoint" as defined by OAuth 2.0 [[RFC6749](#)]. It uses "bearer token" as defined by [[RFC6750](#)].

In addition to the terms defined in referenced specifications, this document uses the following terms:

OAuth: In this document, "OAuth" refers to "OAuth 2.0" [[RFC6749](#)] [[RFC6750](#)].

Browser-based application: An application that is dynamically downloaded and executed in a web browser, usually written in JavaScript. Also sometimes referred to as a "single-page application" or "SPA".

This document discusses the security of browser-based applications, which are executed by the browser in a runtime environment. In most scenarios, these applications are JavaScript applications running in a JavaScript execution environment. Given the popularity of this scenario, this document uses the term "JavaScript" to refer to all mechanisms that allow code to execute in the application's runtime in the browser. The recommendations and considerations in this document are not exclusively linked to the JavaScript language or its runtime but also apply to other languages and runtime environments in the browser, such as WebAssembly (WASM) [[W3C.wasm-core-2](#)].

PKCE: Proof Key for Code Exchange [[RFC7636](#)]. A mechanism that prevents various attacks on OAuth authorization codes.

DPoP: Demonstrating Proof of Possession [[RFC9449](#)]. An OAuth 2.0 mechanism that restricts access tokens for use only by the client they were issued to.

CORS: Cross-Origin Resource Sharing [[Fetch](#)]. A mechanism that enables exceptions to the browser's same-origin policy.

CSP: Content Security Policy [[W3C.CSP3](#)]. A mechanism that restricts which resources a particular web page can fetch or execute.

4. History of OAuth 2.0 in Browser-Based Applications

At the time that OAuth 2.0 was initially specified in [RFC6749] and [RFC6750], browser-based JavaScript applications needed a solution that strictly complied with the same-origin policy. Common deployments of OAuth 2.0 involved an application running on a different domain than the authorization server, so it was historically not possible to use the Authorization Code grant type (Section 4.1 of [RFC6749]), which would require a cross-origin POST request. This limitation was one of the motivations for the definition of the Implicit grant (Section 4.2 of [RFC6749]), which returns the access token in the front channel via the fragment part of the URL, bypassing the need for a cross-origin POST request.

However, there are several drawbacks to the Implicit grant, generally involving vulnerabilities associated with the exposure of the access token in the URL. See Section 7.2 for an analysis of these attacks and the drawbacks of using the Implicit grant in browsers. Additional attacks and security considerations can be found in [RFC9700].

In modern web development, widespread adoption of Cross-Origin Resource Sharing (CORS) [Fetch] (which enables exceptions to the same-origin policy) allows browser-based applications to use the OAuth 2.0 Authorization Code flow and make a POST request to exchange the authorization code for an access token at the token endpoint. Since the Authorization Code grant type enables the use of refresh tokens, this behavior has been adopted for browser-based clients as well, even though these clients are still public clients (defined in Section 2.1 of [RFC6749]) with limited to no access to secure storage. Furthermore, adding Proof Key for Code Exchange (PKCE) [RFC7636] to the flow prevents authorization code injection as well as ensures that even if an authorization code is intercepted, it is unusable by an attacker.

For this reason, and from other lessons learned, the current best practice for browser-based applications is to use the OAuth 2.0 Authorization Code grant type with PKCE. There are various architectural patterns for deploying browser-based applications, both with and without a corresponding server-side component. Each of these architectures has specific trade-offs and considerations that are discussed further in this document. Additional considerations apply for first-party common-domain applications.

5. The Threat of Malicious JavaScript

Malicious JavaScript poses a significant risk to browser-based applications. Attack vectors, such as cross-site scripting (XSS) or the compromise of remote code files, give an attacker the capability to run arbitrary code in the application's execution context. This malicious code is not isolated from the main application's code in any way. Consequently, the malicious code can not only take control of the running execution context but also perform actions within the application's origin. Concretely, this means that the malicious code can steal data from the current page, interact with other same-origin browsing contexts, send requests to a backend from within the application's origin, steal data from origin-based storage mechanisms (e.g., localStorage, IndexedDB), etc.

First and foremost, it is crucial to take proactive measures to avoid the attacker from gaining a foothold in the first place. Doing so involves, but is not limited to, the following:

- Strictly applying context-sensitive output encoding and sanitization when handling untrusted data
- Limiting or avoiding the loading of unchecked third-party resources
- Using Subresource Integrity [[W3C.SRI](#)] to restrict valid scripts that can be loaded
- Using a nonce-based or hash-based Content Security Policy (CSP) [[W3C.CSP3](#)] to prevent the execution of unauthorized script code
- Using origin isolation and HTML5 sandboxing to create boundaries between different parts of the application

Further recommendations can be found in the Open Worldwide Application Security Project (OWASP) Cheat Sheet series [[OWASPCheatSheet](#)].

Unfortunately, history shows that even when applying these security guidelines, there is a risk that the attacker might find a way to trigger the execution of malicious JavaScript. When analyzing the security of browser-based applications in light of the presence of malicious JavaScript, it is crucial to realize that the **malicious JavaScript code has the same privileges as the legitimate application code**. All JavaScript applications are exposed to this risk in some degree.

Applications might obtain OAuth tokens that confer authorization necessary for them to function. In combination, this effectively gives compromised code the ability to use that authorization for malicious ends. Though the risk of attacker abuse of authorization is unavoidable, there are ways to limit the extent to which a compromised application can abuse that authorization. For instance, this access might be limited to times when the application is in active use, by limiting the type of tokens that might be obtained or by binding the tokens to the browser.

When the legitimate application code can access variables or call functions, the malicious JavaScript code can do exactly the same. Furthermore, the malicious JavaScript code can tamper with the regular execution flow of the application, as well as with any application-level defenses, since they are typically controlled from within the application. For example, the attacker can remove or override event listeners, modify the behavior of built-in functions (prototype pollution), and stop pages in frames from loading.

The impact of malicious JavaScript on browser-based applications is a widely studied and well-understood topic. However, the concrete impact of malicious JavaScript on browser-based applications acting as an OAuth client is quite unique, since the malicious JavaScript can now impact the interactions during an OAuth flow. This section explores the threats malicious JavaScript code poses to a browser-based application with the responsibilities of an OAuth client. [Section 5.1](#) discusses a few scenarios that attackers can use once they have found a way to run malicious JavaScript code. These scenarios paint a clear picture of the true power of the attacker, which goes way beyond simple token exfiltration. [Section 5.2](#) analyzes the impact of these attack scenarios on the OAuth client.

The remainder of this specification will refer back to these attack scenarios and consequences to analyze the security properties of the different architectural patterns.

5.1. Attack Scenarios

This section presents several attack scenarios that an attacker can execute once they have found a vulnerability that allows the execution of malicious JavaScript code. The attack scenarios include trivial scenarios ([Section 5.1.1](#)) and elaborate scenarios ([Section 5.1.3](#)). Note that this enumeration is non-exhaustive, narrowly scoped to OAuth-specific features, and presented in no particular order.

5.1.1. Single-Execution Token Theft

The single-execution token theft scenario covers a simple token exfiltration attack, where the attacker obtains and exfiltrates the client's current tokens. This scenario consists of the following steps:

- Execute malicious JavaScript code
- Obtain tokens from the application's preferred storage mechanism ([Section 8](#))
- Send the tokens to a server controlled by the attacker
- Store or abuse the stolen tokens

The recommended defensive strategy to decrease the risk associated with compromised access tokens is to reduce the scope and lifetime of the token. For refresh tokens, the use of refresh token rotation (as defined in [Section 4.14.2](#) of [RFC9700]) offers a detection and correction mechanism. Sender-constrained tokens ([Section 9.2](#)) offer an additional layer of protection against stolen access tokens.

Note that this attack scenario is trivial and often used to illustrate the dangers of malicious JavaScript. When discussing the security of browser-based applications, it is crucial to avoid limiting the attacker's capabilities to the attack discussed in this scenario.

5.1.2. Persistent Token Theft

The persistent token theft attack scenario is a more advanced variation on the single-execution token theft scenario ([Section 5.1.1](#)). Instead of immediately stealing tokens upon the execution of the malicious code, the attacker sets up the necessary handlers to steal the application's tokens on a continuous basis. This scenario consists of the following steps:

- Execute malicious JavaScript code
- Set up a continuous token theft mechanism (e.g., on a 10-second time interval)
 - Obtain tokens from the application's preferred storage mechanism ([Section 8](#))
 - Send the tokens to a server controlled by the attacker
 - Store the tokens
- Wait until the opportune moment to abuse the latest version of the stolen tokens

The crucial difference in this scenario is that the attacker always has access to the latest tokens used by the application. This slight variation in the attack scenario already suffices to counter typical defenses against token theft, such as short lifetimes or refresh token rotation.

For access tokens, the attacker now obtains the latest access token for as long as the user's browser is online. Refresh token rotation is not sufficient to prevent abuse of a refresh token. An attacker can easily ensure that the application will not use the latest refresh token. For example, the attacker could clear the application's tokens after stealing them, wait until the user closes the application, or wait until the user's browser goes offline. Since the application will not use the latest refresh token, there will be no detectable refresh token reuse, giving the attacker full control over the stolen refresh token.

5.1.3. Acquisition and Extraction of New Tokens

In the advanced acquisition and extraction of new tokens attack scenario, the attacker completely disregards any tokens that the application has already obtained. Instead, the attacker takes advantage of the ability to run malicious code that is associated with the application's origin. With that ability, the attacker can inject a hidden iframe and launch a silent Authorization Code flow. This silent flow will reuse the user's existing session with the authorization server and result in the issuing of a new, independent access token (and optionally refresh token). This scenario consists of the following steps:

- Execute malicious JavaScript code
- Set up a handler to obtain the authorization code from the iframe (e.g., by monitoring the frame's URL or via Web Messaging [[WebMessaging](#)])
- Insert a hidden iframe into the page and initialize it with an authorization request. The authorization request in the iframe will occur within the user's session and, if the session is still active, result in the issuing of an authorization code. Note that this step relies on the authorization server supporting silent frame-based flows, as discussed in the last paragraph of this scenario.
- Extract the authorization code from the iframe using the previously installed handler
- Send the authorization code to a server controlled by the attacker
- Exchange the authorization code for a new set of tokens
- Abuse the stolen tokens

The most important takeaway from this scenario is that it runs a new OAuth flow instead of focusing on stealing existing tokens. In essence, even if the application finds a token storage mechanism that is able to completely isolate the stored tokens from the attacker, the attacker will still be able to request a new set of tokens. Note that because the attacker controls the application in the browser, the attacker's Authorization Code flow is indistinguishable from a legitimate Authorization Code flow.

This attack scenario is possible because the security of public browser-based OAuth clients relies entirely on the redirect URI and application's origin. When the attacker executes malicious JavaScript code in the application's origin, they gain the capability to inspect same-origin frames. As a result, the attacker's code running in the main execution context can inspect the redirect URI loaded in the same-origin frame to extract the authorization code.

There are no practical security mechanisms for frontend applications that counter this attack scenario. Short access token lifetimes and refresh token rotation are ineffective, since the attacker has a fresh, independent set of tokens. Advanced security mechanism, such as DPOP [RFC9449], are equally ineffective, since the attacker can use their own key pair to set up and use DPOP for the newly obtained tokens. Requiring user interaction with every Authorization Code flow would effectively stop the automatic silent issuance of new tokens, but this would significantly impact widely established patterns, such as bootstrapping an application on its first page load or single sign-on across multiple related applications, and is not a practical measure.

5.1.4. Proxying Requests via the User's Browser

This attack scenario involves the attacker sending requests to the OAuth resource server directly from within the OAuth client application running in the user's browser. In this scenario, there is no need for the attacker to abuse the application to obtain tokens, since the browser will include its own cookies or tokens in the request. The requests sent by the attacker to the resource server are indistinguishable from requests sent by the legitimate application, since the attacker is running code in the same context as the legitimate application. This scenario consists of the following steps:

- Execute malicious JavaScript code
- Send a request to a resource server and process the response

To authorize the requests to the resource server, the attacker simply mimics the behavior of the client application. For example, when a client application programmatically attaches an access token to outgoing requests, the attacker does the same. Should the client application rely on an external component to augment the request with the proper access token, then this external component will also augment the attacker's request.

This attack pattern is well known and also occurs with server-side web applications using `HttpOnly` session cookies. It is commonly accepted that this scenario cannot be stopped or prevented by application-level security measures. For example, DPOP [RFC9449] explicitly considers this attack scenario to be out of scope.

5.2. Attack Consequences

Successful execution of an attack scenario can result in the theft of access tokens and refresh tokens or in the ability to hijack the client application running in the user's browser. Each of these consequences is relevant for browser-based OAuth clients. They are discussed below in decreasing order of severity.

5.2.1. Exploiting Stolen Refresh Tokens

When the attacker obtains a valid refresh token from a browser-based OAuth client, they can abuse the refresh token by running a Refresh Token grant with the authorization server. The response of the Refresh Token grant contains an access token, which gives the attacker the ability to access protected resources (see [Section 5.2.2](#)). In essence, abusing a stolen refresh token enables long-term impersonation of the legitimate client application in requests to resource servers.

The attack is only stopped when the authorization server refuses a refresh token because it has expired or rotated or when the refresh token is revoked. In a typical browser-based OAuth client, it is not uncommon for a refresh token to remain valid for multiple hours or even days.

5.2.2. Exploiting Stolen Access Tokens

If the attacker obtains a valid access token, they gain the ability to impersonate the legitimate client application in a request to a resource server. Concretely, possession of an access token allows the attacker to send arbitrary requests to any resource server that accepts the valid access token. In essence, abusing a stolen access token enables short-term impersonation of the legitimate client application to resource servers.

The attack ends when the access token expires or when a token is revoked with the authorization server. In a typical browser-based OAuth client, access token lifetimes can be quite short, ranging from minutes to hours.

Note that the possession of the access token allows its unrestricted use by the attacker. The attacker can send arbitrary requests to resource servers using any HTTP method, destination URL, header values, or body.

The application can use DPoP to ensure its access tokens are bound to non-exportable keys held by the browser. In that case, it becomes significantly harder for the attacker to abuse stolen access tokens. More specifically, with DPoP, the attacker can only abuse stolen application tokens by carrying out an online attack where the proofs are calculated in the user's browser. This attack is described in detail in [Section 11.4](#) of [\[RFC9449\]](#). However, when the attacker obtains a fresh access token (and optionally refresh token), as described in [Section 5.1.3](#), they can set up DPoP for these tokens using an attacker-controlled key pair. In that case, the attacker is again free to abuse this newly obtained access token without restrictions.

5.2.3. Client Hijacking

When stealing tokens is not possible or desirable, the attacker can also choose to hijack the OAuth client application running in the user's browser. This effectively allows the attacker to perform any operations that the legitimate client application can perform. Examples include inspecting data on the page, modifying the page, and sending requests to backend systems. Alternatively, the attacker can also abuse their access to the application to launch additional attacks, such as tricking the client into acting on behalf of the attacker using an attack such as session fixation [[SessionFixation](#)].

Note that client hijacking is less powerful than directly abusing stolen user tokens. In a client-hijacking scenario, the attacker cannot directly control the tokens and is restricted by the security policies enforced on the client application. For example, a resource server running on `admin.example.org` can be configured with a CORS policy that rejects requests coming from a client running on `web.example.org`. Even if the access token used by the client would be accepted by the resource server, the resource server's strict CORS configuration would not allow such a request. A resource server without such a strict CORS policy can still be subject to adversarial requests coming from the compromised client application.

6. Application Architecture Patterns

There are three main architectural patterns available when building browser-based applications that rely on OAuth for accessing protected resources:

- A browser-based application that relies on a backend component for handling OAuth responsibilities and forwards all requests through the backend component ("Backend for Frontend" or "BFF")
- A browser-based application that relies on a backend component for handling OAuth responsibilities but calls resource servers directly using the access token (token-mediating backend)
- A browser-based application acting as the client, handling all OAuth responsibilities in the browser (browser-based OAuth client)

Each of these architectural patterns offers a different trade-off between security and simplicity. The patterns in this section are presented in decreasing order of security.

6.1. Backend for Frontend (BFF)

This section describes the architecture of a browser-based application that relies on a backend component to handle all OAuth responsibilities and API interactions. The BFF has three core responsibilities:

1. The BFF interacts with the authorization server as a confidential OAuth client (as defined in [Section 2.1](#) of [\[RFC6749\]](#)).
2. The BFF manages OAuth access and refresh tokens in the context of a cookie-based session, avoiding the direct exposure of any tokens to the browser-based application.
3. The BFF forwards all requests to a resource server, augmenting them with the correct access token before forwarding them to the resource server.

In this architecture, the BFF runs as a server-side component, but it is a component of the frontend application. To avoid confusion with other architectural concepts, such as API gateways and reverse proxies, it is important to keep in mind that the BFF becomes the OAuth client for the frontend application.

If an attacker is able to execute malicious code within the browser-based application, the application architecture is able to withstand most of the attack scenarios discussed before. Since tokens are only available to the BFF, there are no tokens available to extract from the browser (see "Single-Execution Token Theft" ([Section 5.1.1](#)) and "Persistent Token Theft" ([Section 5.1.2](#))). The BFF is a confidential client, which prevents the attacker from running a new flow within the browser (see "Acquisition and Extraction of New Tokens" ([Section 5.1.3](#))). Since the malicious browser-based code still runs within the application's origin, the attacker is able to send requests to the BFF from within the user's browser (see "Proxying Requests via the User's Browser" ([Section 5.1.4](#))). Note that the use of `HttpOnly` cookies prevents the attacker from directly accessing the session state, which prevents the escalation from client hijacking to session hijacking.

6.1.1. Application Architecture

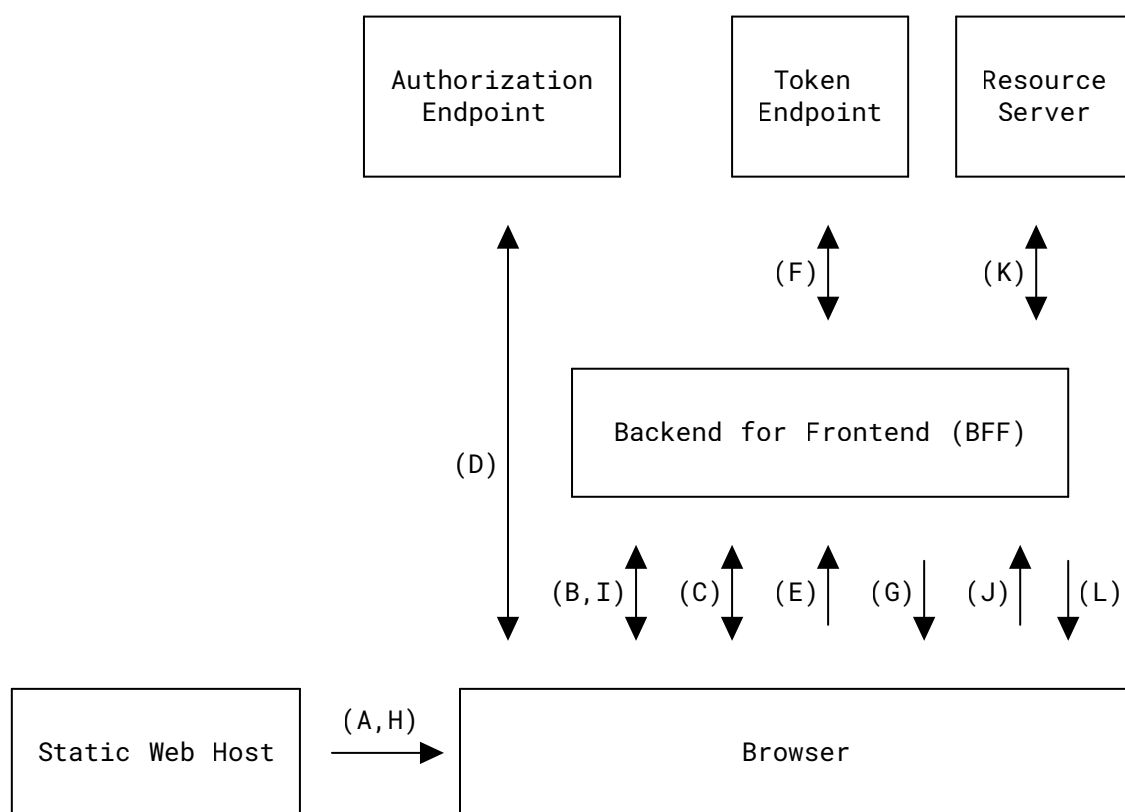


Figure 1: OAuth 2.0 BFF Pattern

In this architecture, the browser code (typically JavaScript) is first loaded from a static web host into the browser (A), and the application then runs in the browser. The application checks with the BFF to identify if there is an active session by calling a "check session" API endpoint (B). If an active session is found, the application resumes its authenticated state and skips forward to (J).

When no active session is found, the browser-based application triggers a navigation to the BFF (C) to initiate the Authorization Code flow with the PKCE extension (described in [Section 6.1.3.1](#)), to which the BFF responds by redirecting the browser to the authorization endpoint (D). When the user is redirected back, the browser delivers the authorization code to the BFF (E), where the BFF can then exchange it for tokens at the token endpoint (F) using its client credentials and PKCE code verifier.

The BFF associates the obtained tokens with the user's session (see [Section 6.1.2.3](#)) and sets a cookie in the response to keep track of this session (G). At this point, the redirect-based Authorization Code flow has been completed, so the BFF can hand control back to the frontend application. It does so by including a redirect in the response (G), triggering the browser to fetch the frontend from the server (H). Note that step (H) is identical to step (A), which likely means that the requested resources can be loaded from the browser's cache. When the frontend loads, it will check with the BFF for an existing session (I), allowing the application to resume its authenticated state.

When the application in the browser wants to make a request to the resource server, it sends a request to the corresponding endpoint on the BFF (J). This request will include the cookie set in (G), allowing the BFF to obtain the proper tokens for this user's session. The BFF removes the cookie from the request, attaches the user's access token to the request, and forwards it to the actual resource server (K). The BFF then forwards the response back to the browser-based application (L).

6.1.2. Implementation Details

6.1.2.1. Session and OAuth Endpoints

The BFF provides a set of endpoints that are crucial to implement the interactions between the browser-based application and the BFF. This section discusses these endpoints in more detail to clarify their purpose and use cases.

The "check session" endpoint ((B) and (I) in [Figure 1](#)) is an API endpoint called by the browser-based application. The request will carry session information when available, allowing the BFF to check for an active session. The response should indicate to the browser-based application whether the session is active. Additionally, the BFF can include other information, such as identity information about the authenticated user.

The endpoint that initiates the Authorization Code flow (C) is contacted by the browser through a navigation. When the application detects an unauthenticated state after checking the session (B), it can navigate the browser to this endpoint. Doing so allows the BFF to respond with a redirect, which takes the browser to the authorization server. The endpoint to initiate this flow is typically included as the "login" endpoint by libraries that support OAuth 2.0 for confidential clients running on a web server. Note that it is also possible for the BFF to initiate the Authorization Code flow in (B) when it detects the absence of an active session. In that case, the BFF would return the authorization URI in the response and expect the application to trigger a navigation event with this URI. However, this scenario requires a custom implementation and makes it harder to use standard OAuth libraries.

The endpoint that receives the authorization code (E) is called by a navigation event from within the browser. At this point, the application is not loaded and not in a position to handle the redirect. Similar to the initiation of the flow, the endpoint to handle the redirect is offered by standard OAuth libraries. The BFF can respond to this request with a redirect that triggers the browser to load the application.

Finally, the BFF can also offer a "logout" endpoint to the application, which is not depicted in [Figure 1](#). The exact behavior of the logout endpoint depends on the application requirements. Note that standard OAuth libraries typically also offer an implementation of the "logout" endpoint.

6.1.2.2. Refresh Tokens

When using refresh tokens, as described in [Section 4.14](#) of [\[RFC9700\]](#), the BFF obtains the refresh token (F) and associates it with the user's session.

If the BFF notices that the user's access token has expired and the BFF has a refresh token, it can use the refresh token to obtain a fresh access token. Since the BFF OAuth client is a confidential client, it will use client authentication on the refresh token request. Typically, the BFF performs these steps inline when handling an API call from the frontend. In that case, these steps, which are not explicitly shown in [Figure 1](#), would occur between (J) and (K). BFFs that keep all token information available on the server side can also request fresh access tokens when they observe a token expiration event to increase the performance of API requests.

When the refresh token expires, there is no way to obtain a valid access token without running an entirely new Authorization Code flow. Therefore, it makes sense to configure the lifetime of the cookie-based session managed by the BFF to be equal to the maximum lifetime of the refresh token. Additionally, when the BFF learns that a refresh token for an active session is no longer valid, it also makes sense to invalidate the session.

6.1.2.3. Cookie-Based Session State

The BFF relies on browser cookies [\[RFC6265bis\]](#) to keep track of the user's session, which is used to access the user's tokens. Cookie-based sessions, both server-side and client-side, have some downsides.

Server-side sessions expose only a session identifier and keep all data on the server. Doing so ensures a great level of control over active sessions, along with the possibility to revoke any session at will. The downside of this approach is the impact on scalability, requiring solutions such as "sticky sessions" or "session replication". Given these downsides, using server-side sessions with a BFF is only recommended in small-scale scenarios.

Client-side sessions push all data to the browser in a signed, and optionally encrypted, object. This pattern absolves the server of keeping track of any session data, but it severely limits control over active sessions and makes it difficult to handle session revocation. However, when client-side sessions are used in the context of a BFF, these properties change significantly. Since the cookie-based session is only used to obtain a user's tokens, all control and revocation properties

follow from the use of access tokens and refresh tokens. It suffices to revoke the user's access token and/or refresh token to prevent ongoing access to protected resources, without the need to explicitly invalidate the cookie-based session.

Best practices to secure the session cookie are discussed in [Section 6.1.3.2](#).

6.1.2.4. Combining OAuth and OpenID Connect

The OAuth flow used by this application architecture can be combined with OpenID Connect by including the necessary OpenID Connect scopes in the authorization request (C) (at least the scope `openid` as defined in Section 3.1.2.1 of [\[OpenID\]](#)). In that case, the BFF will receive an ID Token in (F). The BFF can associate the information from the ID Token with the user's session and provide it to the application in (B) or (I).

When needed, the BFF can use the access token associated with the user's session to make requests to the UserInfo endpoint.

6.1.2.5. Practical Deployment Strategies

Serving the static JavaScript code is a separate responsibility from handling OAuth tokens and forwarding requests. In [Figure 1](#), the BFF and static web host are shown as two separate entities. In real-world deployments, these components can be deployed as a single service (i.e., the BFF serving the static JavaScript code), as two separate services (i.e., a Content Delivery Network (CDN) and a BFF), or as two components in a single service (i.e., static hosting and serverless functions on a cloud platform).

Note that it is possible to further customize this architecture to support specific scenarios. For example, an application relying on both internal and external resource servers can choose to host the internal resource server alongside the BFF. In that scenario, requests to the internal resource server are handled directly at the BFF, without the need to forward requests over the network. Authorization from the point of view of the resource server does not change, as the user's session is internally translated to the access token and its claims.

6.1.3. Security Considerations

6.1.3.1. The Authorization Code Grant

The main benefit of using a BFF is the BFF's ability to act as a confidential client. Therefore, the BFF **MUST** act as a confidential client by establishing credentials with the authorization server. Furthermore, the BFF **MUST** use the OAuth 2.0 Authorization Code grant as described in [Section 2.1.1](#) of [\[RFC9700\]](#) to initiate a request for an access token.

6.1.3.2. Cookie Security

The BFF uses cookies to create a user session, which is directly associated with the user's tokens through either server-side or client-side session state. Given the sensitive nature of these cookies, they must be properly protected.

The following cookie security guidelines are relevant for this particular BFF architecture:

- The BFF **MUST** enable the Secure flag for its cookies.
- The BFF **MUST** enable the HttpOnly flag for its cookies.
- The BFF **SHOULD** enable the SameSite=Strict flag for its cookies.
- The BFF **SHOULD** set its cookie path to /.
- The BFF **SHOULD NOT** set the Domain attribute for cookies.
- The BFF **SHOULD** start the name of its cookie with a prefix indicating the cookie was set via HTTP, for example, by using the __Host-Http- prefix defined in [\[COOKIES\]](#).

Note: In new deployments, all of the above requirements are likely to be straightforward to implement. The "SHOULD" items are not deemed as "MUST" items so that existing architectures can be compliant. The implications of these requirements are listed below.

These cookie security guidelines, combined with the use of HTTPS, help counter attacks that directly target a cookie-based session. Session hijacking is not possible due to the Secure and HttpOnly cookie flags. The __Host-Http- prefix prevents the cookie from being shared with subdomains, thereby countering subdomain-based session hijacking or session-fixation attacks. In a typical BFF deployment scenario, there is no reason to use more relaxed cookie security settings than the requirements listed above. Deviating from these settings requires proper motivation for the deployment scenario at hand.

Additionally, when using client-side sessions that contain access tokens (as opposed to server-side sessions where the tokens only live on the server), the BFF **SHOULD** encrypt its cookie contents. While the use of cookie encryption does not affect the security properties of the BFF pattern, it does ensure that tokens stored in cookies are never written to the user's local persistent storage in plaintext format. This security measure helps ensure the confidentiality of the tokens in case an attacker is able to read cookies from the hard drive. Such an attack can be launched through malware running on the victim's computer. Note that while encrypting the cookie contents prevents direct access to embedded tokens, it still allows the attacker to use the encrypted cookie in a session hijacking attack.

For further guidance on cookie security best practices, refer to the OWASP Cheat Sheet series [\[OWASPCheatSheet\]](#).

6.1.3.3. Cross-Site Request Forgery Protections

The interactions between the browser-based application and the BFF rely on cookies for authentication and authorization. Similar to other cookie-based interactions, the BFF is required to account for Cross-Site Request Forgery (CSRF) attacks. A successful CSRF attack could allow the attacker's request to the BFF to trigger outgoing calls to a protected resource.

The BFF **MUST** implement a proper CSRF defense. The exact mechanism or combination of mechanisms depends on the exact domain where the BFF is deployed, as discussed below.

6.1.3.3.1. SameSite Cookie Attribute

Configuring the cookies with the SameSite=Strict attribute (see [Section 6.1.3.2](#)) ensures that the BFF's cookies are only included on same-site requests and not on potentially malicious cross-site requests.

This defense is adequate if the BFF is never considered to be same site with any other applications. However, it falls short when the BFF is hosted alongside other applications within the same site, defined as the eTLD+1 (see this definition in [\[Site\]](#) for more details).

For example, subdomains such as `https://a.example.com` and `https://b.example.com` are considered to be same site because they share the same site `example.com`. They are considered to be cross-origin because origins consist of the tuple `<scheme, hostname, port>`. As a result, a subdomain-takeover attack against `b.example.com` can enable CSRF attacks against the BFF of `a.example.com`. Note that these subdomain-based attacks follow the same pattern as CSRF attacks but with a cross-origin nature instead of a cross-site nature.

6.1.3.3.2. Cross-Origin Resource Sharing

The BFF can rely on CORS as a CSRF defense mechanism. CORS is a security mechanism implemented by browsers that restrict cross-origin requests, unless the server explicitly approves such a request by setting the proper CORS headers.

Browsers typically restrict cross-origin HTTP requests initiated from scripts. CORS can remove this restriction if the target server approves the request, which is checked through an initial "preflight" request. Unless the preflight response explicitly approves the request, the browser will refuse to send the full request.

Because of this property, the BFF can rely on CORS as a CSRF defense. When the attacker tries to launch a cross-origin request to the BFF from the user's browser, the BFF will not approve the request in the preflight response, causing the browser to block the actual request. Note that the attacker can always launch the request from their own machine, but then the request will not carry the user's cookies, so the attack will fail.

When relying on CORS as a CSRF defense, it is important to realize that certain requests are possible without a preflight. For such requests, called "CORS-safelisted requests", the browser will simply send the request and prevent access to the response if the server did not send the proper CORS headers. This behavior is enforced for requests that can be triggered via other means than JavaScript, such as a GET request or a form-based POST request.

The consequence of this behavior is that certain endpoints of the resource server could become vulnerable to CSRF, even with CORS enabled as a defense. For example, if the resource server is an API that exposes an endpoint to a bodiless POST request, there will be no preflight request and no CSRF defense.

To avoid such bypasses against the CORS policy, the BFF **SHOULD** require that the browser-based application includes a custom request header. Cross-origin requests with a custom request header always require a preflight, which makes CORS an effective CSRF defense. When this

mechanism is used, the BFF **MUST** ensure that every incoming request carries this static header. The exact naming of this header is at the discretion of the application and BFF. A sample configuration would be a request header with a static value, such as `My-Static-Header: 1`.

It is also possible to deploy the browser-based application on the same origin as the BFF. This ensures that legitimate interactions between the frontend and the BFF do not require any preflights, so there's no additional overhead.

6.1.3.3.3. Use Anti-Forgery/Double-Submit Cookies

Some technology stacks and frameworks have built-in CSRF protection using anti-forgery cookies. This mechanism relies on a session-specific secret that is stored in a cookie, which can only be read by the legitimate frontend running in the domain associated with the cookie. The frontend is expected to read the cookie and insert its value into the request, typically by adding a custom request header. The backend verifies the value in the cookie to the value provided by the frontend to identify legitimate requests. When implemented correctly for all state-changing requests, this mechanism effectively mitigates CSRF.

Note that this mechanism is not necessarily recommended over the CORS approach. However, if a framework offers built-in support for this mechanism, it can serve as a low-effort alternative to protect against CSRF.

6.1.3.4. Privacy Considerations in the BFF Architecture

The BFF pattern requires that the browser-based application forwards all requests to a resource server through a backend BFF component. As a consequence, the BFF component is able to observe all requests and responses between the application and a resource server, which can have a considerable privacy impact.

When the browser-based application and BFF are built and deployed by the same party, the privacy impact is likely minimal. However, when this pattern is implemented using a BFF component that is provided or hosted by a third party, this privacy impact needs to be taken into account.

6.1.3.5. Operational Considerations

As the BFF is forwarding all requests to the resource server on behalf of the frontend, care should be taken to ensure the resource server is aware of this component and uses appropriate policies for rate limiting and other anti-abuse measures. For example, if the BFF is deployed as a single-instance service and the resource server is rate limiting requests based on an IP address, the server might start blocking requests as many users' browsers will appear to be coming from the single IP address of the BFF.

6.1.3.6. Proxy Restrictions

The BFF acts as a proxy service by accepting requests from the frontend and forwarding them to the resource server. The inbound request carries a cookie, which the BFF translates into an access token on the outbound request. (Note that this makes it more like an application-layer reverse proxy rather than an HTTP proxy.) Apart from CSRF attacks, attackers may attempt to

manipulate the BFF into forwarding requests to unintended hosts. If an attacker successfully exploits this, they could redirect the BFF to an arbitrary server, potentially exposing the user's access token.

To mitigate this risk, the BFF **MUST** enforce strict outbound request controls by validating destination hosts before forwarding requests. This requires maintaining an explicit allowlist of approved resource servers, ensuring that requests are only proxied to predefined backends (e.g., `/bff/orders/create` maps exclusively to `https://order-api.example.com/create`). If dynamic routing based on paths (e.g., `/bff/orders/{id}`) is necessary, the BFF **MUST** apply strict validation to ensure that only authorized destinations are accessible. Additionally, restricting the allowed HTTP methods on a per-endpoint basis can further reduce attack vectors.

When implementing a dynamically configurable proxy, the BFF **MUST** ensure that it only allows requests to explicitly permitted hosts and paths. Failure to enforce these restrictions can lead to unauthorized access and access token leakage.

6.1.3.7. Advanced Security

In the BFF pattern, all OAuth responsibilities have been moved to the BFF, a server-side component acting as a confidential client. Since server-side applications run in a more controlled environment than browser-based applications, it becomes easier to adopt advanced OAuth security practices. Examples include key-based client authentication and sender-constrained tokens.

6.1.4. Threat Analysis

This section revisits the attack scenarios and consequences from [Section 5](#) and discusses potential additional defenses.

6.1.4.1. Attack Scenarios and Consequences

If the attacker has the ability to execute malicious code (e.g., JavaScript or WASM) in the application's execution context, the following attack scenarios become relevant:

- Proxying Requests via the User's Browser ([Section 5.1.4](#))

Note that this attack scenario results in the following consequences:

- Client Hijacking ([Section 5.2.3](#))

Note that client hijacking is an attack scenario that is inherent to the nature of browser-based applications. As a result, nothing will be able to prevent such attacks apart from stopping the execution of malicious code in the first place. Techniques that can help to achieve this are following secure coding guidelines, code analysis, and deploying defense-in-depth mechanisms such as Content Security Policy [[W3C.CSP3](#)].

In this architecture, the BFF is a key component handling various security-specific responsibilities and proxy-based behavior. While it is out of the scope of this document to discuss a secure implementation of proxy-based applications, it is crucial to note that security vulnerabilities in the BFF can have a significant impact on the application.

Finally, the BFF is uniquely placed to observe all traffic between the browser-based application and the resource servers. If a high-security application would prefer to implement anomaly detection or rate limiting, such a BFF would be the ideal place to do so. Such restrictions can further help to mitigate the consequences of client hijacking.

6.1.4.2. Mitigated Attack Scenarios

The other attack scenarios, listed below, are effectively mitigated by the BFF application architecture:

- Single-Execution Token Theft ([Section 5.1.1](#))
- Persistent Token Theft ([Section 5.1.2](#))
- Acquisition and Extraction of New Tokens ([Section 5.1.3](#))

The BFF counters the first two attack scenarios by not exposing any tokens to the browser-based application. Even when the attacker gains full control over the application, there are simply no tokens to be stolen.

The third scenario, where the attacker obtains a fresh access token (and optionally refresh token) by running a silent flow, is mitigated by making the BFF a confidential client. Even when the attacker manages to obtain an authorization code, they are prevented from exchanging this code due to the lack of client credentials. Additionally, the use of PKCE prevents other attacks against the authorization code.

Since refresh and access tokens are managed by the BFF and not exposed to the browser, the following two consequences of potential attacks become irrelevant:

- Exploiting Stolen Refresh Tokens ([Section 5.2.1](#))
- Exploiting Stolen Access Tokens ([Section 5.2.2](#))

6.1.4.3. Summary

The architecture of a BFF is significantly more complicated than a browser-only application. It requires deploying and operating a server-side BFF component. Additionally, this pattern requires all interactions between the application and the resource servers to be proxied by the BFF. Depending on the deployment pattern, this proxy behavior can add a significant burden on the server-side components. See [Section 6.2.2.6](#) for additional notes if the BFF is acting as the resource server.

However, because of the nature of the BFF architecture pattern, it offers strong security guarantees. Using a BFF also ensures that the application's attack surface does not increase by using OAuth. The only viable attack pattern is hijacking the client application in the user's browser -- a problem inherent to web applications.

This architecture is strongly recommended for business applications, sensitive applications, and applications that handle personal data.

6.2. Token-Mediating Backend

This section describes the architecture of a browser-based application that relies on a backend component to handle OAuth responsibilities for obtaining tokens as a confidential client (as defined in [Section 2.1](#) of [\[RFC6749\]](#)). The backend component then provides the application with the access token to directly interact with resource servers.

The token-mediating backend pattern is more lightweight than the BFF pattern (see [Section 6.1](#)), since it does not require the proxying of all requests and responses between the application and the resource server. From a security perspective, the token-mediating backend is less secure than a BFF but still offers significant advantages over an OAuth client application running directly in the browser.

If an attacker is able to execute malicious code within the application, the application architecture is able to prevent the attacker from abusing refresh tokens (see "Single-Execution Token Theft" ([Section 5.1.1](#)) and "Persistent Token Theft" ([Section 5.1.2](#))) or obtaining a fresh set of tokens (see "Acquisition and Extraction of New Tokens" ([Section 5.1.3](#))). However, since the access token is directly exposed to the application, the attacker can steal the token from client-side storage (see "Single-Execution Token Theft" ([Section 5.1.1](#)) and "Persistent Token Theft" ([Section 5.1.2](#))) or request a fresh token from the token-mediating backend (see "Proxying Requests via the User's Browser" ([Section 5.1.4](#))). Note that the use of `HttpOnly` cookies prevents the attacker from directly accessing the session state, which prevents the escalation from access token theft to session hijacking.

6.2.1. Application Architecture

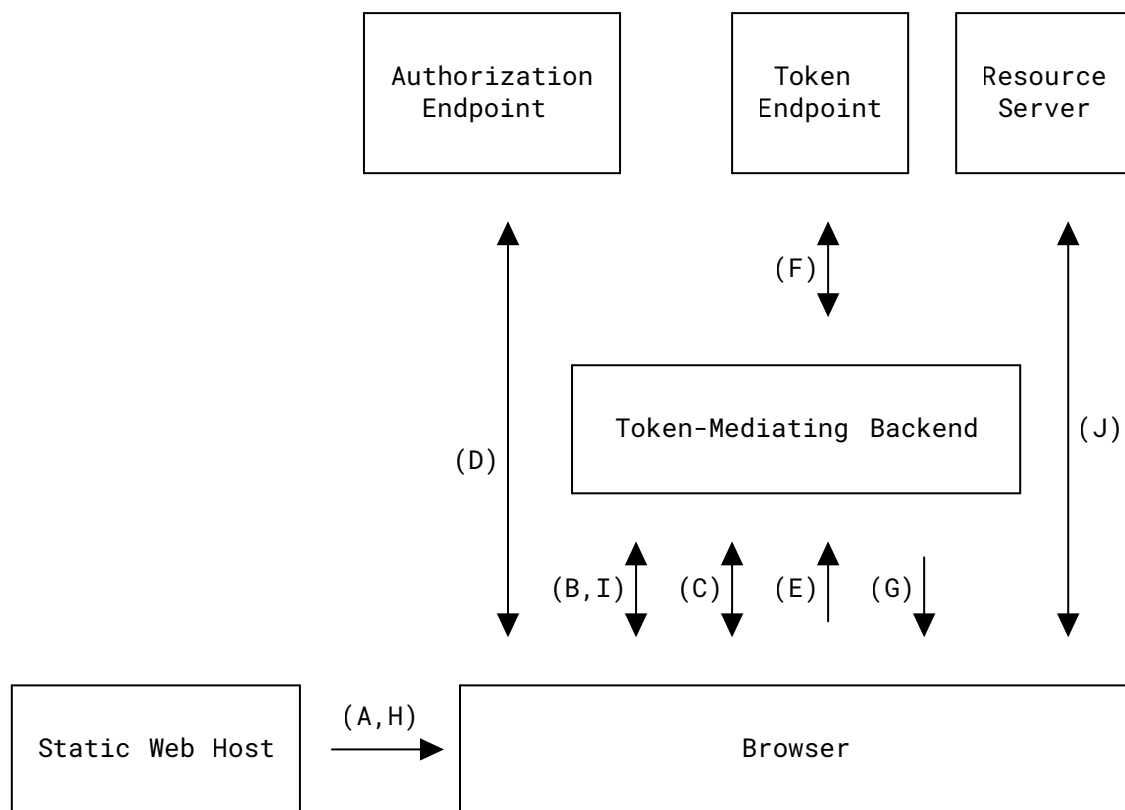


Figure 2: OAuth 2.0 Token-Mediating Backend Pattern

In this architecture, the browser-based code (e.g., JavaScript or WASM) is first loaded from a static web host into the browser (A), and the application then runs in the browser. The application checks with the token-mediating backend if there is an active session (B). If an active session is found, the application receives the corresponding access token, resumes its authenticated state, and skips forward to (J).

When no active session is found, the application triggers a navigation to the token-mediating backend (C) to initiate the Authorization Code flow with the PKCE extension (described in [Section 6.2.3.1](#)), to which the token-mediating backend responds by redirecting the browser to the authorization endpoint (D). When the user is redirected back, the browser delivers the authorization code to the token-mediating backend (E), where the token-mediating backend can then exchange it for tokens at the token endpoint (F) using its client credentials and PKCE code verifier.

The token-mediating backend associates the obtained tokens with the user's session (see [Section 6.2.2.4](#)) and sets a cookie in the response to keep track of this session (G). This response to the browser will also trigger the reloading of the application (H). When this application reloads, it will check with the token-mediating backend for an existing session (I), allowing the application to resume its authenticated state and obtain the access token from the token-mediating backend.

The application in the browser can use the access token obtained in (I) to directly make requests to the resource server (J).

6.2.2. Implementation Details

6.2.2.1. Session and OAuth Endpoints

Most of the endpoint implementations of the token-mediating backend are similar to those described for a BFF.

- The "check session" endpoint ((B) and (I) in [Figure 2](#)) is an API endpoint called by the browser-based application. The request will carry session information when available, allowing the backend to check for an active session. The response should indicate to the browser-based application whether the session is active. If an active session is found, the backend includes the access token in the response. Additionally, the backend can include other information, such as identity information about the authenticated user.
- The endpoint that initiates the Authorization Code flow (C) is identical to the endpoint described for the BFF architecture. See [Section 6.1.2.1](#) for more details.
- The endpoint that receives the authorization code (E) is identical to the endpoint described for the BFF architecture. See [Section 6.1.2.1](#) for more details.
- The endpoint that supports logout is identical to the endpoint described for the BFF architecture. See [Section 6.1.2.1](#) for more details.

6.2.2.2. Refresh Tokens

When using refresh tokens, as described in [Section 4.14](#) of [\[RFC9700\]](#), the token-mediating backend obtains the refresh token in (F) and associates it with the user's session.

If the resource server rejects the access token, the application can contact the token-mediating backend to request a new access token. The token-mediating backend relies on the cookies associated with this request to look up the user's refresh token and makes a token request using the refresh token. These steps are not shown in [Figure 2](#). Note that this refresh token request is from the backend, a confidential client, and thus requires client authentication.

When the refresh token expires, there is no way to obtain a valid access token without starting an entirely new Authorization Code grant. Therefore, it makes sense to configure the lifetime of the cookie-based session to be equal to the maximum lifetime of the refresh token if such information is known upfront. Additionally, when the token-mediating backend learns that a refresh token for an active session is no longer valid, it makes sense to invalidate the session.

6.2.2.3. Access Token Scopes

Depending on the resource servers being accessed and the configuration of scopes at the authorization server, the application may wish to request access tokens with different scope configurations. This behavior would allow the application to follow the best practice of using minimally scoped access tokens.

The application can inform the token-mediating backend of the desired scopes when it checks for the active session ((A) and (I)). It is up to the token-mediating backend to decide if previously obtained access tokens fall within the desired scope criteria.

It should be noted that this access token caching mechanism at the token-mediating backend can cause scope-elevation risks when applied indiscriminately. If the cached access token features a superset of the scopes requested by the frontend, the token-mediating backend **SHOULD NOT** return it to the frontend; instead, it **SHOULD** use the refresh token to request an access token with the smaller set of scopes from the authorization server. Note that support of such an access token downscoping mechanism is at the discretion of the authorization server.

The token-mediating backend can use a similar mechanism to the downscoping mechanism when relying on the extension described in [\[RFC8707\]](#) to obtain an access token for a specific resource server.

6.2.2.4. Cookie-Based Session State

Similar to the BFF, the token-mediating backend relies on browser cookies to keep track of the user's session. The implementation guidelines and security considerations for cookie-based sessions are the same as those for a BFF, as discussed in [Section 6.1.2.3](#).

6.2.2.5. Combining OAuth and OpenID Connect

Similar to a BFF, the token-mediating backend can choose to combine OAuth and OpenID Connect in a single flow. See [Section 6.1.2.4](#) for more details.

6.2.2.6. Practical Deployment Scenarios

Hosting the static JavaScript or WASM code is a separate responsibility from handling interactions with the authorization server. In [Figure 2](#), the token-mediating backend and static web host are shown as two separate entities. In real-world deployment scenarios, these components can be deployed as a single service (i.e., the token-mediating backend serving the static code), as two separate services (i.e., a CDN and a token-mediating backend), or as two components in a single service (i.e., static hosting and serverless functions on a cloud platform). These deployment differences do not affect the relationships described in this pattern but may impact other practicalities, such as the need to properly configure CORS to enable cross-origin communication.

6.2.3. Security Considerations

6.2.3.1. The Authorization Code Grant

The main benefit of using a token-mediating backend is the backend's ability to act as a confidential client. Therefore, the token-mediating backend **MUST** act as a confidential client. Furthermore, the token-mediating backend **MUST** use the OAuth 2.0 Authorization Code grant as described in [Section 2.1.1](#) of [\[RFC9700\]](#) to initiate a request for an access token.

6.2.3.2. Cookie Security

The token-mediating backend uses cookies to create a user session, which is directly associated with the user's tokens through either server-side or client-side session state. The same cookie security guidelines established for a BFF also apply, as discussed in [Section 6.1.3.2](#).

6.2.3.3. Cross-Site Request Forgery Protections

Interactions between the browser-based application and the token-mediating backend rely on cookies for authentication and authorization. Just like a BFF, the token-mediating backend is required to account for CSRF attacks.

[Section 6.1.3.3](#) outlines the nuances of various mitigation strategies against CSRF attacks. Specifically for a token-mediating backend, these CSRF defenses only apply to the endpoint or endpoints where the application can obtain its access tokens.

6.2.3.4. Advanced OAuth Security

The token-mediating backend is a confidential client running as a server-side component. The token-mediating backend can adopt security best practices for confidential clients, such as key-based client authentication.

6.2.4. Threat Analysis

This section revisits the attack scenarios and consequences from [Section 5](#) and discusses potential additional defenses.

6.2.4.1. Attack Scenarios and Consequences

If the attacker has the ability to execute malicious code in the application's execution context, the following attack scenarios become relevant:

- Single-Execution Token Theft ([Section 5.1.1](#)) for access tokens
- Persistent Token Theft ([Section 5.1.2](#)) for access tokens
- Proxying Requests via the User's Browser ([Section 5.1.4](#))

Note that these attack scenarios result in the following consequences:

- Exploiting Stolen Access Tokens ([Section 5.2.2](#))
- Client Hijacking ([Section 5.2.3](#))

Exposing the access token to the browser-based application is the core idea behind the architecture pattern of the token-mediating backend. As a result, the access token becomes vulnerable to token theft by malicious browser-based code.

6.2.4.2. Mitigated Attack Scenarios

The other attack scenarios (listed below) are effectively mitigated by the token-mediating backend:

- Single-Execution Token Theft ([Section 5.1.1](#)) for refresh tokens

- Persistent Token Theft ([Section 5.1.2](#)) for refresh tokens
- Acquisition and Extraction of New Tokens ([Section 5.1.3](#))

The token-mediating backend counters the first two attack scenarios by not exposing the refresh token to the browser-based application. Even when the attacker gains full control over the application, there are simply no refresh tokens to be stolen.

The third scenario, where the attacker obtains a fresh access token (and optionally refresh token) by running a silent flow, is mitigated by making the token-mediating backend a confidential client. Even when the attacker manages to obtain an authorization code, they are prevented from exchanging this code due to the lack of client credentials. Additionally, the use of PKCE prevents other attacks against the authorization code.

Because of the nature of the token-mediating backend, the following consequences of potential attacks become irrelevant:

- Exploiting Stolen Refresh Tokens ([Section 5.2.1](#))

6.2.4.3. Additional Defenses

While this architecture inherently exposes access tokens, there are some additional defenses that can help to increase the security posture of the application.

6.2.4.3.1. Secure Token Storage

Given the nature of the token-mediating backend pattern, there is no need for persistent token storage in the browser. When needed, the application can always use its cookie-based session to obtain an access token from the token-mediating backend. [Section 8](#) provides more details on the security properties of various storage mechanisms in the browser.

Be aware that even when the access token is stored out of reach of malicious browser-based code, the malicious code can still mimic the legitimate application and send a request to the token-mediation backend to obtain the latest access token.

6.2.4.3.2. Using Sender-Constrained Tokens

Using sender-constrained access tokens is not trivial in this architecture. The token-mediating backend is responsible for exchanging an authorization code or refresh token for an access token, but the application will use the access token. Using a mechanism such as DPoP [[RFC9449](#)] would require splitting responsibilities over two parties, which is not a scenario defined by the specification. Use of DPoP in such a scenario is out of the scope of this document.

6.2.4.4. Summary

The architecture of a token-mediating backend is more complicated than a browser-only application but is less complicated than running a proxying BFF. Similarly, the security properties offered by the token-mediating backend lie somewhere between using a BFF and running a browser-only application.

A token-mediating backend addresses typical scenarios that grant the attacker long-term access on behalf of the user. However, due to the consequence of access token theft, the attacker still has the ability to gain direct access to resource servers.

When considering a token-mediating backend architecture, it is strongly recommended to evaluate if adopting a full BFF as discussed in [Section 6.1](#) is a viable alternative. Only when the use cases or system requirements would prevent the use of a proxying BFF should the token-mediating backend be considered over a full BFF.

6.3. Browser-Based OAuth 2.0 Client

This section describes the architecture of a browser-based application that acts as the OAuth client, handling all OAuth responsibilities in the browser. As a result, the browser-based application obtains tokens from the authorization server, without the involvement of a backend component.

If an attacker is able to execute malicious code in the browser, this application architecture is vulnerable to all attack scenarios discussed earlier ([Section 5.1](#)). In essence, the attacker will be able to obtain access tokens and refresh tokens from the authorization server, potentially giving them long-term access to protected resources on behalf of the user.

6.3.1. Application Architecture

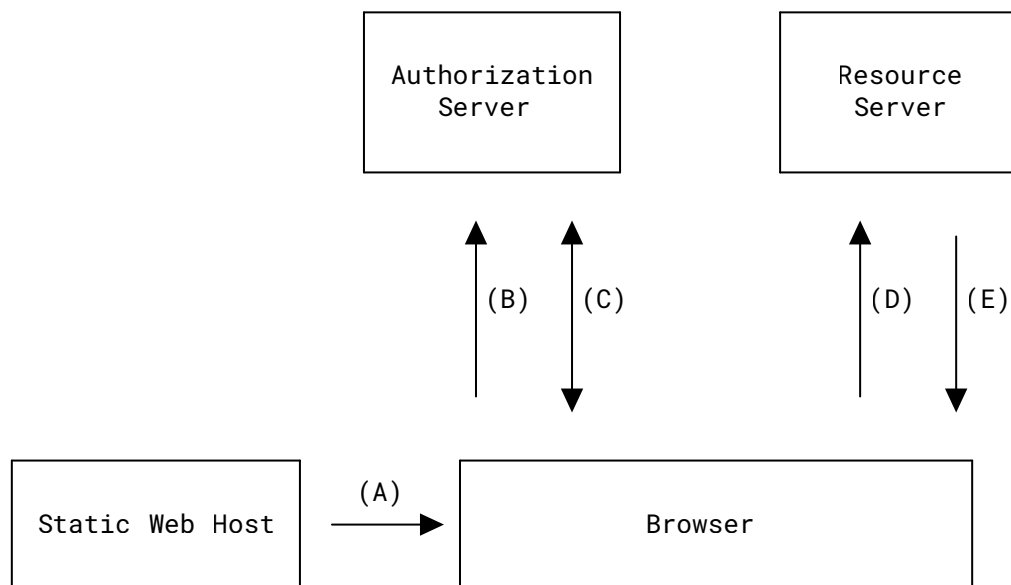


Figure 3: Browser-Based OAuth 2.0 Client Pattern

In this architecture, the code is first loaded from a static web host into the browser (A), and then the application runs in the browser. In this scenario, the browser-based application is considered a public client, which does not possess client credentials to authenticate to the authorization server.

The application obtains an authorization code (B) by initiating the Authorization Code flow with the PKCE extension (described in [Section 6.3.2.1](#)). The application uses a browser API (e.g., [\[Fetch\]](#)) to make a POST request to the token endpoint (C) to exchange the authorization code for tokens.

The application is then responsible for storing the access token and optional refresh token as securely as possible using appropriate browser APIs, as described in [Section 8](#).

When the application in the browser wants to make a request to the resource server, it can interact with the resource server directly. The application includes the access token in the request (D) and receives the resource server's response (E).

6.3.2. Implementation Details

Browser-based applications that are public clients ([Section 2.1](#) of [\[RFC6749\]](#)) and use the Authorization Code grant type described in [Section 4.1](#) of [\[RFC6749\]](#) **MUST** also follow the additional requirements described in this section.

6.3.2.1. The Authorization Code Grant

Browser-based applications that are public clients **MUST** implement the PKCE [\[RFC7636\]](#) extension when obtaining an access token, and authorization servers **MUST** support and enforce PKCE for such clients.

The PKCE extension prevents an attack where the authorization code is intercepted and exchanged for an access token by a malicious client by providing the authorization server with a way to verify that the client instance that exchanges the authorization code is the same one that initiated the flow.

6.3.2.2. Cross-Site Request Forgery Protections

Browser-based applications **MUST** prevent CSRF attacks against their redirect URI. This can be accomplished by any of the below:

- configuring the authorization server to require PKCE for this client
- using and verifying unique value for the OAuth state parameter to carry a CSRF token
- using and verifying the OpenID Connect nonce parameter as described in [\[OpenID\]](#) if the application is also using OpenID Connect

See [Section 2.1](#) of [\[RFC9700\]](#) for additional details on selecting a proper CSRF defense for the Authorization Code flow.

6.3.2.3. Refresh Tokens

For browser-based clients, the refresh token is typically a bearer token unless the application explicitly uses DPoP [RFC9449]. As a result, the risk of leaked refresh tokens is greater than that of leaked access tokens, since an attacker may be able to continue using the stolen refresh token to obtain new access tokens potentially without being detected by the authorization server.

Authorization servers may choose whether or not to issue refresh tokens to browser-based applications. However, in light of the impact of third-party cookie-blocking mechanisms, the use of refresh tokens has become significantly more attractive. [RFC9700] describes some additional requirements around refresh tokens on top of the recommendations of [RFC6749]. Applications and authorization servers conforming to this BCP **MUST** also follow the recommendations in [RFC9700] for refresh tokens if refresh tokens are issued to browser-based applications.

In particular, authorization servers:

- **MUST** either rotate refresh tokens on each use OR use sender-constrained refresh tokens as described in Section 4.14.2 of [RFC9700]
- **MUST** either set a maximum lifetime on refresh tokens OR expire if the refresh token has not been used within some amount of time
- **MUST NOT**, upon issuing a rotated refresh token, extend the lifetime of the new refresh token beyond the lifetime of the initial refresh token if the refresh token has a preestablished expiration time

Limiting the overall refresh token lifetime to the lifetime of the initial refresh token ensures a stolen refresh token cannot be used indefinitely.

For example:

- A user authorizes an application, issuing an access token that lasts 10 minutes and a refresh token that lasts 8 hours.
- After 10 minutes, the initial access token expires, so the application uses the refresh token to get a new access token.
- The authorization server returns a new access token that lasts 10 minutes and a new refresh token that lasts 7 hours and 50 minutes.
- This continues until 8 hours pass from the initial authorization.
- At this point, when the application attempts to use the refresh token after 8 hours, the request will fail and the application will have to re-initiate an Authorization Code flow that relies on the user's authentication or previously established session.

Authorization servers **SHOULD** link the lifetime of the refresh token to the user's authenticated session with the authorization server. Doing so ensures that when a user logs out, previously issued refresh tokens to browser-based applications become invalid, mimicking a single-logout scenario. Authorization servers **MAY** set different policies for refresh token issuance, lifetime, and expiration for browser-based applications compared to other public clients.

6.3.3. Security Considerations

6.3.3.1. Client Authentication

Since a browser-based application's source code is delivered to the end-user's browser, it is unfit to contain provisioned secrets. As a consequence, browser-based applications are typically deployed as public clients, as defined by [Section 2.1](#) of [\[RFC6749\]](#).

Secrets that are statically included as part of an app distributed to multiple users should not be treated as confidential secrets, as one user may inspect their copy and learn the shared secret. For this reason, and those stated in [Section 5.3.1](#) of [\[RFC6819\]](#), authorization servers **MUST NOT** require client authentication of browser-based applications using a shared secret, as this serves no value beyond client identification, which is already provided by the `client_id` parameter.

Authorization servers that still require a statically included shared secret for SPA clients **MUST** treat the client as a public client and not accept the secret as proof of the client's identity. Without additional measures, such clients are subject to client impersonation (see [Section 6.3.3.2](#)).

6.3.3.2. Client Impersonation

As stated in [Section 10.2](#) of [\[RFC6749\]](#), the authorization server **SHOULD NOT** process authorization requests automatically without user consent or interaction, except when the authorization server can assure the identity of the client application.

If authorization servers restrict redirect URIs to a fixed set of absolute HTTPS URIs that prevent the use of wildcard domains, wildcard paths, or wildcard query string components, this exact match of registered absolute HTTPS URIs **MAY** be accepted by authorization servers as proof of identity of the client for the purpose of deciding whether to automatically process an authorization request when a previous request for the `client_id` has already been approved.

6.3.3.2.1. Authorization Code Redirect

Clients **MUST** register one or more redirect URIs with the authorization server and use only exact registered redirect URIs in the authorization request.

Authorization servers **MUST** require an exact match of a registered redirect URI as described in [Section 4.1.1](#) of [\[RFC9700\]](#). This helps to prevent attacks targeting the authorization code.

6.3.3.3. Security of In-Browser Communication Flows

In browser-based applications, it is common to execute the OAuth flow in a secondary window, such as a pop-up or iframe, instead of redirecting the primary window. In these flows, the browser-based app holds control of the primary window, for instance, to avoid page refreshes or to run frame-based flows silently.

If the browser-based app and the authorization server are invoked in different frames, they have to use in-browser communication techniques like the `postMessage` API (a.k.a. [\[WebMessaging\]](#)) instead of top-level redirections. To guarantee confidentiality and authenticity of messages, both the initiator origin and receiver origin of a `postMessage` **MUST** be verified using the mechanisms inherently provided by the `postMessage` API (Section 9.3.2 in [\[WebMessaging\]](#)).

[Section 4.17](#) of [\[RFC9700\]](#) provides additional details about the security of in-browser communication flows and the countermeasures that browser-based applications and authorization servers **MUST** apply to defend against these attacks.

6.3.3.4. Cross-Origin Requests

In this scenario, the application uses a browser API to send requests to the authorization server and the resource server. Given the nature of OAuth 2.0, these requests are typically cross-origin requests, subjecting them to browser-enforced restrictions on cross-origin communication. The authorization server and the resource server **MUST** send necessary CORS headers (defined in [\[Fetch\]](#)) to enable the application to make the necessary cross-origin requests. Note that in the extraordinary scenario where the browser-based OAuth client runs in the same origin as the authorization server or resource server, a CORS policy is not needed to enable the necessary interaction.

For the authorization server, the CORS configuration is relevant for the token endpoint, where the browser-based application exchanges the authorization code for tokens. Additionally, if the authorization server provides additional endpoints to the application such as discovery metadata URLs, JSON Web Key Sets, dynamic client registration, revocation, introspection, or user info endpoints, these endpoints may also be accessed by the browser-based application. Consequently, the authorization server is responsible for supporting CORS on these endpoints.

This specification does not include guidelines for deciding the concrete CORS policy implementation, which can consist of a wildcard origin or a more restrictive configuration. Note that CORS has two modes of operation with different security properties. The first mode applies to CORS-safelisted requests (formerly known as simple requests), where the browser sends the request and uses the CORS response headers to decide if the response can be exposed to the client-side execution context. For non-CORS-safelisted requests, such as a request with a custom request header, the browser will first check the CORS policy using a preflight. The browser will only send the actual request when the server sends its approval in the preflight response.

Note that due to the authorization server's specific configuration, it is possible that the CORS response to a preflight is different from the CORS response to the actual request. During the preflight, the authorization server can only verify the provided origin, but during an actual request, the authorization server has the full request data, such as the client ID. Consequently, the authorization server can approve a known origin during the preflight but reject the actual request after comparing the origin to this specific client's list of pre-registered origins.

6.3.4. Threat Analysis

This section revisits the attack scenarios and consequences from [Section 5](#) and discusses potential additional defenses.

6.3.4.1. Attack Scenarios and Consequences

If the attacker has the ability to execute malicious code in the application's execution context, the following attack scenarios become relevant:

- Single-Execution Token Theft ([Section 5.1.1](#))
- Persistent Token Theft ([Section 5.1.2](#))
- Acquisition and Extraction of New Tokens ([Section 5.1.3](#))
- Proxying Requests via the User's Browser ([Section 5.1.4](#))

The most dangerous attack scenario is the acquisition and extraction of new tokens. In this attack scenario, the attacker only interacts with the authorization server, which makes the actual implementation details of the OAuth functionality in the client irrelevant. Even if the legitimate client application finds a way to completely isolate the tokens from the attacker, the attacker will still be able to obtain tokens from the authorization server.

Note that these attack scenarios result in the following consequences:

- Exploiting Stolen Refresh Tokens ([Section 5.2.1](#))
- Exploiting Stolen Access Tokens ([Section 5.2.2](#))
- Client Hijacking ([Section 5.2.3](#))

6.3.4.2. Additional Defenses

While this architecture is inherently vulnerable to malicious browser-based code, there are some additional defenses that can help to increase the security posture of the application. Note that none of these defenses address or fix the underlying problem that allows the attacker to run a new flow to obtain tokens.

6.3.4.2.1. Secure Token Storage

When handling tokens directly, the application can choose different storage mechanisms to store access tokens and refresh tokens. Universally accessible storage areas, such as *Local Storage* [[WebStorage](#)], are easier to access from malicious JavaScript than more isolated storage areas, such as a *Web Worker* [[WebWorker](#)]. [Section 8](#) discusses different storage mechanisms with their trade-off in more detail.

A practical implementation pattern can use a Web Worker [[WebWorker](#)] to isolate the refresh token and provide the application with the access token making requests to resource servers. This prevents an attacker from using the application's refresh token to obtain new tokens.

However, even a token storage mechanism that completely isolates the tokens from the attacker does not prevent the attacker from running a new flow to obtain a fresh set of tokens (see [Section 5.1.3](#)).

6.3.4.2.2. Using Sender-Constrained Tokens

Browser-based OAuth clients can implement DPoP [RFC9449] to transition from bearer access tokens and bearer refresh tokens to sender-constrained tokens. In such an implementation, the private key used to sign DPoP proofs is handled by the browser (a non-extractable [CryptoKeyPair] is stored using [W3C.IndexedDB]). As a result, the use of DPoP effectively prevents scenarios where the XSS attacker exfiltrates the application's tokens (see Sections 5.1.1 and 5.1.2).

Note that the use of DPoP does not prevent the attacker from running a new flow to obtain a fresh access token (and optionally a refresh token); see Section 5.1.3. Even when DPoP is mandatory, the attacker can bind the fresh set of tokens to a key pair under their control, allowing them to exfiltrate the sender-constrained tokens and use them by relying on the attacker-controlled key to calculate the necessary DPoP proofs.

6.3.4.2.3. Restricting Access to the Authorization Server

The scenario where the attacker obtains a fresh access token and (optionally refresh token) Section 5.1.3 relies on the ability to directly interact with the authorization server from within the browser. In theory, a defense that prevents the attacker from silently interacting with the authorization server could solve the most dangerous attack scenario. However, in practice, such defenses are ineffective or impractical.

For completeness, this BCP lists a few options below. Note that none of these defenses is recommended, as they do not offer practically usable security benefits.

The authorization server could block authorization requests that originate from within an iframe. While this would prevent the exact scenario from Section 5.1.3, it would not work for slight variations of the attack scenario. For example, the attacker can launch the silent flow in a pop-up window or a pop-under window. Additionally, browser-only OAuth clients typically rely on a hidden iframe-based flow to bootstrap the user's authentication state, so this approach would significantly impact the user experience.

The authorization server could opt to make user consent mandatory in every Authorization Code flow (as described in Section 10.2 of [RFC6749]), thus requiring user interaction before issuing an authorization code. This approach would make it harder for an attacker to run a silent flow to obtain a fresh set of tokens. However, it also significantly impacts the user experience by continuously requiring consent. As a result, this approach would result in "consent fatigue", which makes it likely that the user will blindly approve the consent, even when it is associated with a flow that was initiated by the attacker.

6.3.4.3. Summary

To summarize, the architecture of a browser-based OAuth client application is straightforward but results in a significant increase in the attack surface of the application. The attacker is not only able to hijack the client but is also able to extract a full-featured set of tokens from the browser-based application.

This architecture is not recommended for business applications, sensitive applications, and applications that handle personal data.

7. Discouraged and Deprecated Architecture Patterns

Client applications and backend applications have evolved significantly over the last two decades, along with threats, attacker models, and a general understanding of modern application security. As a result, previous recommendations generally accepted in the industry as well as published by the OAuth Working Group are often no longer recommended, and proposed solutions often fall short of meeting the expected security requirements.

This section discusses a few alternative architecture patterns, which are not recommended for use in modern browser-based OAuth applications. This section discusses each of the patterns along with a threat analysis that investigates the attack scenarios and consequences when relevant.

7.1. Single-Domain Browser-Based Applications (Not Using OAuth)

Too often, simple applications are made needlessly complex by using OAuth to replace the concept of session management. A typical example is the modern incarnation of a server-side Model-View-Controller (MVC) application, which now consists of a browser-based frontend backed by a server-side API.

In such an application, the use of OpenID Connect to offload user authentication to a dedicated provider can significantly simplify the application's architecture and development. However, the use of OAuth for governing access between the frontend and the backend is often not needed. Instead of using access tokens, the application can rely on server-side cookie-based session state to keep track of the user's authentication status. The security guidelines to protect the session cookie are discussed in [Section 6.1.3.2](#).

While the advice to not use OAuth seems out of place in this document, it is important to note that OAuth was originally created for third-party or federated access to APIs, so it may not be the best solution in a single common-domain deployment. That said, there are still some advantages in using OAuth even in a common-domain architecture, such as:

- Allowing more flexibility in the future. For example, if you were to later add a new domain to the system, having OAuth already in place means that a new domain wouldn't require any additional rearchitecting.
- Being able to take advantage of existing library support rather than writing bespoke code for the integration.
- Centralizing login and multi-factor authentication support, account management, and recovery at the OAuth server rather than making it part of the application logic.
- Splitting responsibilities between authenticating a user and serving resources.

Using OAuth for browser-based applications in a first-party same-domain scenario provides these advantages and can be accomplished by any of the architectural patterns described above.

7.1.1. Threat Analysis

Due to the lack of using OAuth, this architecture pattern is only vulnerable to the following attack scenario: proxying requests via the user's browser ([Section 5.1.4](#)). As a result, this pattern can lead to the following consequence: client hijacking ([Section 5.2.3](#)).

7.2. OAuth Implicit Grant

The OAuth 2.0 Implicit grant type (defined in [Section 4.2](#) of [\[RFC6749\]](#)) works by the authorization server issuing an access token in the authorization response (front channel) without an authorization code exchange step. In this case, the access token is returned in the fragment part of the redirect URI, providing an attacker with several opportunities to intercept and steal the access token.

The security properties of the Implicit grant type make it no longer a recommended best practice. To effectively prevent the use of this flow, the authorization server **MUST NOT** issue access tokens in the authorization response and **MUST** issue access tokens only from the token endpoint. Browser-based clients **MUST** use the Authorization Code grant type and **MUST NOT** use the Implicit grant type to obtain access tokens.

7.2.1. Historic Note

Historically, the Implicit grant type provided an advantage to browser-based applications since JavaScript could always arbitrarily read and manipulate the fragment portion of the URL without triggering a page reload. This was necessary in order to remove the access token from the URL after it was obtained by the app. Additionally, until CORS was widespread in browsers, the Implicit grant type offered an alternative flow that didn't require CORS support in the browser or on the server.

Modern browsers now have the Session History API (described in "Navigation and session history" of [\[HTML\]](#)), which provides a mechanism to modify the path and query string component of the URL without triggering a page reload. Additionally, CORS has widespread support and is often used by single-page applications for many purposes. This means modern browser-based applications can use the OAuth 2.0 Authorization Code grant type with PKCE, since they have the ability to remove the authorization code from the query string without triggering a page reload thanks to the Session History API, and CORS support at the token endpoint means the app can obtain tokens even if the authorization server is on a different domain.

7.2.2. Threat Analysis

The architecture pattern discussed in this section is vulnerable to the following attack scenarios:

- Single-Execution Token Theft ([Section 5.1.1](#))
- Persistent Token Theft ([Section 5.1.2](#))
- Acquisition and Extraction of New Tokens ([Section 5.1.3](#))
- Proxying Requests via the User's Browser ([Section 5.1.4](#))

As a result, this pattern can lead to the following consequences:

- Exploiting Stolen Refresh Tokens ([Section 5.2.1](#))
- Exploiting Stolen Access Tokens ([Section 5.2.2](#))
- Client Hijacking ([Section 5.2.3](#))

7.2.3. Further Attacks on the Implicit Grant

Apart from the attack scenarios and consequences that were already discussed, there are a few additional attacks that further support the deprecation of the Implicit grant type. Many attacks on the Implicit grant type described by [\[RFC6819\]](#) and [Section 4.1.2](#) of [\[RFC9700\]](#) do not have sufficient mitigation strategies. The following sections describe the specific attacks that cannot be mitigated while continuing to use the Implicit grant type.

7.2.3.1. Manipulation of the Redirect URI

If an attacker is able to cause the authorization response to be sent to a URI under their control, they will get access directly to the authorization response including the access token. Several methods of performing this attack are described in detail in [\[RFC9700\]](#).

7.2.3.2. Access Token Leak in Browser History

An attacker could obtain the access token from the browser's history. The countermeasures recommended in [\[RFC6819\]](#) are limited to using short expiration times for tokens and indicate that browsers should not cache the response. Neither of these fully prevent this attack; they only reduce the potential damage.

Additionally, many browsers now also sync browser history to cloud services and to multiple devices, providing an even wider attack surface to extract access tokens out of the URL.

This is discussed in more detail in [Section 4.3.2](#) of [\[RFC9700\]](#).

7.2.3.3. Manipulation of Scripts

An attacker could modify the page or inject scripts into the browser through various means, including when the browser's HTTPS connection is being intercepted by, for example, a corporate network. While attacks on the TLS layer are typically out of scope of basic security recommendations to prevent, in the case of browser-based applications, they are much easier to perform. An injected script can enable an attacker to have access to everything on the page.

The risk of a malicious script running on the page may be amplified when the application uses a known standard way of obtaining access tokens; namely, the attacker can always look at the `window.location` variable to find an access token. This threat profile is different from an attacker specifically targeting an individual application by knowing where or how an access token obtained via the Authorization Code flow may end up being stored.

7.2.3.4. Access Token Leak to Third-Party Scripts

It is relatively common to use third-party scripts in browser-based applications, such as analytics tools, crash reporting, and even things like a social media "like" button. In these situations, the author of the application may not be able to be fully aware of the entirety of the code running in the application. When an access token is returned in the fragment, it is visible to any third-party scripts on the page.

7.2.4. Disadvantages of the Implicit Grant

There are several additional reasons the Implicit grant type is disadvantageous compared to using the recommended Authorization Code grant type.

- OAuth 2.0 provides no mechanism for a client to verify that a particular access token was intended for that client, which could lead to misuse and possible impersonation attacks if a malicious party hands off an access token it retrieved through some other means to the client.
- Returning an access token in the front-channel redirect gives the authorization server no assurance that the access token will actually end up at the application, since there are many ways this redirect may fail or be intercepted.
- Supporting the Implicit grant type requires additional code, more upkeep and understanding of the related security considerations. Limiting the authorization server to just the Authorization Code grant type reduces the attack surface of the implementation.
- If the browser-based application gets wrapped into a native app, then [\[RFC8252\]](#) also requires the use of the Authorization Code grant type with PKCE anyway.

7.3. Resource Owner Password Credentials Grant

The Resource Owner Password Credentials Grant **MUST NOT** be used, as described in [Section 2.4](#) of [\[RFC9700\]](#). Instead, using the Authorization Code grant type and redirecting the user to the authorization server provides the authorization server the opportunity to prompt the user for secure non-phishable authentication options, take advantage of single sign-on sessions, or use third-party identity providers. In contrast, the Resource Owner Password Credentials Grant does not provide any built-in mechanism for these and would instead need to be extended with custom protocols.

To conform to this best practice, browser-based applications using OAuth or OpenID Connect **MUST** use a redirect-based flow (e.g., the OAuth Authorization Code grant type) as described in this document.

7.4. Handling the OAuth Flow in a Service Worker

In an attempt to limit the attacker's ability to extract existing tokens or acquire a new set of tokens, a pattern using a Service Worker [\[W3C.service-workers\]](#) has been suggested in the past. In this pattern, the application's first action upon loading is registering a Service Worker. The Service Worker becomes responsible for executing the Authorization Code flow to obtain tokens and to augment outgoing requests to the resource server with the proper access token.

Additionally, the Service Worker blocks the client application's code from making direct calls to the authorization server's endpoints. This restriction aims to target the attack scenario "Acquisition and Extraction of New Tokens" ([Section 5.1.3](#)).

The sequence diagram included below illustrates the interactions between the client, the Service Worker, the authorization server, and the resource server.

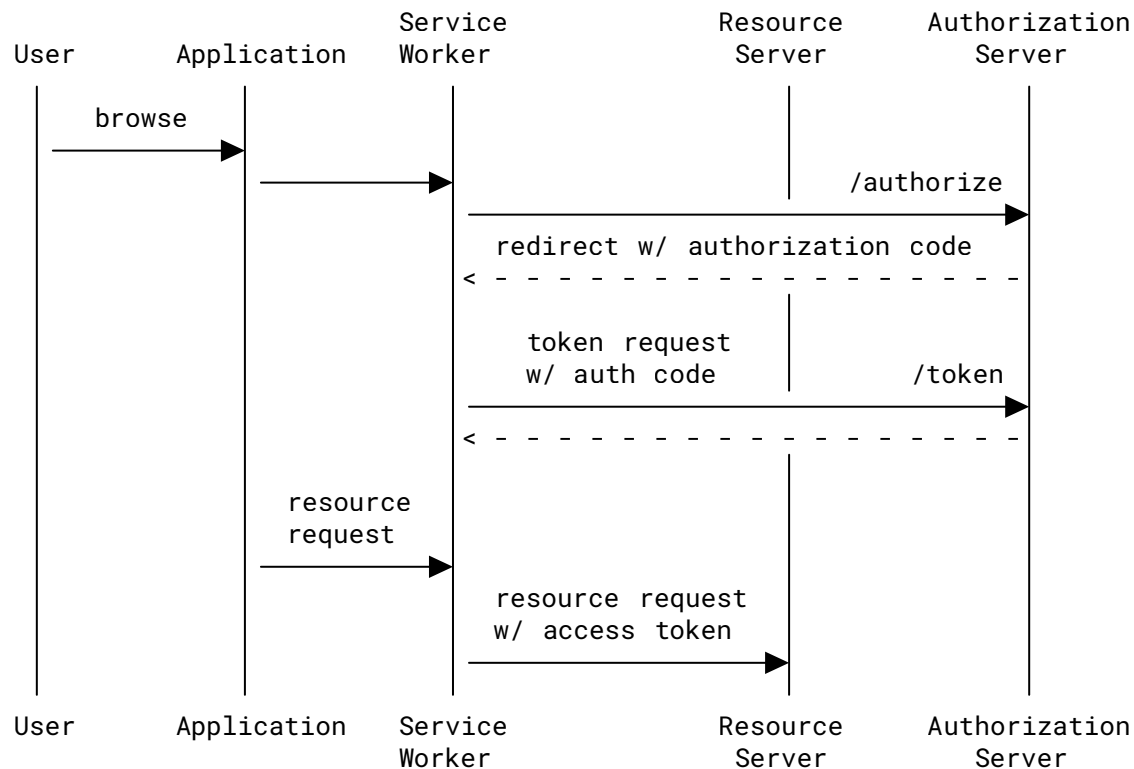


Figure 4: OAuth 2.0 Service Worker Pattern

Note that this pattern never exposes the tokens to the application running in the browser. Since the Service Worker runs in an isolated execution environment, there is no shared memory and no way for the client application to influence the execution of the Service Worker.

7.4.1. Threat Analysis

The architecture pattern discussed in this section is vulnerable to the following attack scenarios:

- Acquisition and Extraction of New Tokens ([Section 5.1.3](#))
- Proxying Requests via the User's Browser ([Section 5.1.4](#))

As a result, this pattern can lead to the following consequences:

- Exploiting Stolen Refresh Tokens ([Section 5.2.1](#))
- Exploiting Stolen Access Tokens ([Section 5.2.2](#))
- Client Hijacking ([Section 5.2.3](#))

7.4.1.1. Attacking the Service Worker

The seemingly promising security benefits of using a Service Worker warrant a more detailed discussion of its security limitations. To fully protect the application against the relevant attack scenarios ([Section 5.1](#)), the Service Worker needs to meet two security requirements:

1. Prevent an attacker from exfiltrating tokens.
2. Prevent an attacker from acquiring a new set of tokens.

Once registered, the Service Worker runs an Authorization Code flow and obtains the tokens. Since the Service Worker keeps track of tokens in its own isolated execution environment, they are out of reach for any application code, including potentially malicious code. Consequently, the Service Worker meets the first requirement of preventing token exfiltration. This essentially neutralizes the first two attack scenarios discussed in [Section 5.1](#).

To meet the second security requirement, the Service Worker must be able to guarantee that an attacker controlling the legitimate application cannot execute a new Authorization Code grant; this attack is discussed in [Section 5.1.3](#). Due to the nature of Service Workers, the registered Service Worker will be able to block all outgoing requests that initiate such a new flow, even when they occur in a frame or a new window.

However, the malicious code running inside the application can unregister this Service Worker. Unregistering a Service Worker can have a significant functional impact on the application, so it is not an operation the browser handles lightly. Therefore, an unregistered Service Worker is marked as such, but all currently running instances remain active until their corresponding browsing context is terminated (e.g., by closing the tab or window). So even when an attacker unregisters a Service Worker, it remains active and able to prevent the attacker from reaching the authorization server.

One of the consequences of unregistering a Service Worker is that it will not be present when a new browsing context is opened. So when the attacker first unregisters the Service Worker and then starts a new flow in a frame, there will be no Service Worker associated with the browsing context of the frame. Consequently, the attacker will be able to run its own new Authorization Code grant, extract the authorization code from the frame's URL, and exchange it for tokens. In essence, the Service Worker fails to meet the second security requirement, leaving it vulnerable to the scenario where the attacker acquires a new set of tokens ([Section 5.1.3](#)).

Due to these shortcomings, combined with the significant complexity of registering and maintaining a Service Worker, this pattern is not recommended.

Finally, note that the use of a Service Worker by itself does not increase the attack surface of the application. In practice, Service Workers are often used to retrofit a legacy application with support for including OAuth access tokens on outgoing requests. The Service Worker in these scenarios does not change the security properties of the application but merely simplifies development and maintenance of the application.

8. Token Storage in the Browser

When a browser-based application handles OAuth access tokens or refresh tokens directly, it becomes responsible for ephemerally or persistently storing the tokens. As a consequence, the application needs to decide how to manage the tokens (e.g., in-memory vs. persistent storage) and which steps to take to further isolate the tokens from the main application code. This section discusses a few different storage mechanisms and their properties. These recommendations take into account the unique properties of OAuth tokens, some of which may overlap with general browser security recommendations.

When discussing the security properties of browser-based token storage solutions, it is important to understand the attacker's capabilities when they compromise a browser-based application. Similar to previous discussions, two main attack scenarios should be taken into account:

1. The attacker obtaining tokens from storage.
2. The attacker obtaining tokens from the provider (e.g., the authorization server or the token-mediating backend).

Since the attacker's code becomes indistinguishable from the legitimate application's code, the attacker will always be able to request tokens from the provider in exactly the same way as the legitimate application code. As a result, not even a completely isolated token storage solution can address the dangers of the second threat, where the attacker requests tokens from the provider.

That said, the different security properties of browser-based storage solutions will impact the attacker's ability to obtain existing tokens from storage.

8.1. Cookies

Browser cookies are both a storage mechanism and a transport mechanism. The browser automatically supports both mechanisms through the corresponding request and response headers, resulting in the storage of cookies in the browser and the automatic inclusion of cookies on outgoing requests provided they match the cookie's domain, path, or other properties.

Next to header-based control over cookies, browsers also offer a JavaScript Cookie API to get and set cookies. This Cookie API is often mistaken as an easy way to store data in the browser. In such a scenario, the JavaScript code stores a token in a cookie with the intent to retrieve the token for later inclusion in the Authorization header of an API call. However, since the cookie is associated with the domain of the browser-based application, the browser will also send the cookie containing the token when making a request to the server running on this domain. One example of such a request is the browser loading the application after a previous visit to the application ((A) in [Figure 3](#)).

Because of these unintentional side effects of using cookies for JavaScript-based storage, this practice is **NOT RECOMMENDED**.

Note that this practice is different from the use of cookies in a BFF (discussed in [Section 6.1.3.2](#)), where the cookie is inaccessible to JavaScript and is intended to be sent to the backend.

8.2. Token Storage in a Service Worker

A Service Worker [[W3C.service-workers](#)] offers a fully isolated environment to keep track of tokens. These tokens are inaccessible to the client application, effectively protecting them against exfiltration. To guarantee the security of these tokens, the Service Worker cannot share these tokens with the application. Consequently, whenever the application wants to perform an operation with a token, it has to ask the Service Worker to perform this operation and return the result.

When aiming to isolate tokens from the application's execution context, the Service Worker **MUST NOT** store tokens in any persistent storage API that is shared with the main window. For example, currently, the IndexedDB storage is shared between the browsing context and Service Worker and therefore is not a suitable place for the Service Worker to persist data that should remain inaccessible to the main window. Consequently, the Service Worker currently does not have access to an isolated persistent storage area.

As discussed before, the use of a Service Worker does not prevent an attacker from obtaining a new set of tokens. Similarly, if the application is responsible for obtaining tokens from the authorization server and passing them to a Service Worker for further management, the attacker can perform the same operation as the legitimate application to obtain these tokens.

8.3. Token Storage in a Web Worker

The application can use a Web Worker [[WebWorker](#)], which results in an almost identical scenario as the previous one that relies on a Service Worker. The difference between a Service Worker and a Web Worker is the level of access and its runtime properties. Service Workers can intercept and modify outgoing requests, while Web Workers are just a way to run background tasks. Web Workers are ephemeral and disappear when the browsing context is closed, while Service Workers are persistent services registered in the browser.

The security properties of using a Web Worker are identical to using Service Workers. When tokens are exposed to the application, they become vulnerable. When tokens need to be used, the operation that relies on them has to be carried out by the Web Worker.

One common method to isolate the refresh token is to use Web Workers. In such a scenario, the application starts an Authorization Code flow from a Web Worker. The authorization code from the redirect is forwarded to the Web Worker, which then exchanges it for tokens. The Web Worker keeps the refresh token in memory and sends the access token to the main application. The main application uses the access token as desired. When the application needs to run a refresh token flow, it asks the Web Worker to do so, after which the application obtains a fresh access token.

In this scenario, the application's own refresh token is effectively protected against exfiltration, but the access token is not. Additionally, nothing would prevent an attacker from obtaining their own tokens by running a new Authorization Code flow ([Section 5.1.3](#)).

8.4. In-Memory Token Storage

Another option is keeping tokens in memory without using any persistent storage. Doing so limits the exposure of the tokens to the current execution context only but has the downside of not being able to persist tokens between page loads.

In a JavaScript execution environment, the security of in-memory token storage can be further enhanced by using a closure variable to effectively shield the token from direct access. By using closures, the token is only accessible to the predefined functions inside the closure, such as a function to make a request to the resource server.

While closures work well in simple, isolated environments, they are tricky to secure in a complex environment like the browser's execution environment. For example, a closure relies on a variety of outside functions to execute its operations, such as `toString` or networking APIs. Using prototype poisoning, an attacker can substitute these functions with malicious versions, causing the closure's future operations to use these malicious versions. Inside the malicious function, the attacker can gain access to the function arguments, which may expose the tokens from within the closure to the attacker.

8.5. Persistent Token Storage

The persistent storage APIs currently available in browsers as of this writing are `localStorage` [[WebStorage](#)], `sessionStorage` [[WebStorage](#)], and `IndexedDB` [[W3C.IndexedDB](#)].

`localStorage` persists between page reloads and is shared across all tabs. This storage is accessible to the entire origin and persists longer term. `localStorage` does not protect against unauthorized access from malicious JavaScript, as the attacker would be running code within the same origin, and as such, would be able to read the contents of the `localStorage`. Additionally, `localStorage` is a synchronous API, blocking other JavaScript until the operation completes.

`sessionStorage` is similar to `localStorage`, except that the lifetime of `sessionStorage` is linked to the lifetime of a browser tab. Additionally, `sessionStorage` is not shared between multiple tabs open to pages on the same origin, which slightly reduces the exposure of the tokens in `sessionStorage`.

`IndexedDB` is a persistent storage mechanism like `localStorage`, but it is shared between multiple tabs as well as between the browsing context and Service Workers. Additionally, `IndexedDB` is an asynchronous API, which is preferred over the synchronous `localStorage` API.

Note that the main difference between these patterns is the exposure of the data, but none of these options can fully mitigate token exfiltration when the attacker can execute malicious code in the application's execution environment.

8.6. Filesystem Considerations for Browser Storage APIs

In all cases, as of this writing, there is no guarantee that browser storage is encrypted at rest. This behavior potentially exposes tokens to attackers that have the ability to read files on disk. While such attacks rely on capabilities that are well beyond the scope of browser-based applications, this topic highlights an important attack vector against modern applications. More and more malware is specifically created to crawl users' machines and look for browser profiles to obtain high-value tokens and session cookies, resulting in account-takeover attacks.

While the browser-based application is incapable of mitigating such attacks, the application can mitigate the consequences of such an attack by ensuring data confidentiality using encryption. The [\[W3C.WebCryptoAPI\]](#) provides a mechanism for JavaScript code to generate a secret key, as well as an option for that key to be non-exportable. A JavaScript application could then use this API to encrypt and decrypt tokens before storing them. However, the [\[W3C.WebCryptoAPI\]](#) specification only ensures that the key is not exportable to the browser code, but it does not place any requirements on the underlying storage of the key itself with the operating system. As such, a non-exportable key cannot be relied on as a way to protect against exfiltration from the underlying filesystem.

In order to protect against token exfiltration from the filesystem, the encryption keys would need to be stored somewhere other than the filesystem, such as on a remote server. This introduces new complexity for a purely browser-based app and is out of scope of this document.

9. Security Considerations

9.1. Reducing the Authority of Tokens

A general security best practice in the OAuth world is to minimize the authority associated with access tokens. This best practice is applicable to all the architectures discussed in this specification. Concretely, the following considerations can help minimize the authority of access tokens:

- Reduce the lifetime of access tokens and rely on refresh tokens for access token renewal
- Reduce the scopes or permissions associated with the access tokens
- Restrict access tokens to a single resource using the extension described in [\[RFC8707\]](#)

When OpenID Connect is used, it is important to avoid sensitive information disclosure through the claims in the ID Token. The authorization server **SHOULD NOT** include any ID Token claims that aren't used by the client.

9.2. Sender-Constrained Tokens

As discussed throughout this document, the use of sender-constrained tokens does not solve the security limitations of browser-only OAuth clients. However, when the level of security offered by a token-mediating backend ([Section 6.2](#)) or a browser-only OAuth client ([Section 6.3](#)) suffices

for the use case at hand, sender-constrained tokens can be used to enhance the security of both access tokens and refresh tokens. One method of implementing sender-constrained tokens in a way that is usable from browser-based applications is DPoP [[RFC9449](#)].

When using sender-constrained tokens, the OAuth client has to prove possession of a private key in order to use the token, such that the token isn't usable by itself. If a sender-constrained token is stolen, the attacker wouldn't be able to use the token directly; they would also need to steal the private key. In essence, one could say that using sender-constrained tokens shifts the challenge of securely storing the token to securely storing the private key. Ideally, the application should use a non-exportable private key, such as generating one with the [[W3C.WebCryptoAPI](#)]. With an unencrypted token in the browser storage protected by a non-exportable private key, an XSS attack would not be able to extract the key, so the token would not be usable by the attacker.

If the application is unable to use an API that generates a non-exportable key, the application should take measures to isolate the private key from its own execution context. The techniques for doing so are similar to using a secure token storage mechanism, as discussed in [Section 8](#).

While a non-exportable key is protected from exfiltration from within the JavaScript context, the exfiltration of the underlying private key from the filesystem is still a potential attack vector. At the time of writing, there is no guarantee made in [[W3C.WebCryptoAPI](#)] that a non-exportable key is actually protected by a Trusted Platform Module (TPM) or stored in an encrypted form on disk. Exfiltration of the non-exportable key from the underlying filesystem may still be possible if the attacker can get access to the filesystem of the user's machine, for example, via malware. This effectively makes the potential attack vector equivalent to a session hijacking attack.

9.3. Authorization Server Mix-Up Mitigation

Authorization server mix-up attacks mark a severe threat to every client that supports at least two authorization servers. [Section 4.4](#) of [[RFC9700](#)] provides additional details about mix-up attacks and the countermeasures mentioned above.

9.4. Isolating Applications Using Origins

Many of the web's security mechanisms rely on origins, which are defined as the triple `<scheme, hostname, port>`. For example, browsers automatically isolate browsing contexts with different origins, limit resources to certain origins, and apply CORS restrictions to outgoing cross-origin requests.

Therefore, it is considered a best practice to avoid deploying more than one application in a single origin. An architecture that only deploys a single application in an origin can leverage these browser restrictions to increase the security of the application. Additionally, having a single origin per application makes it easier to configure and deploy security measures such as CORS, CSP, etc.

10. IANA Considerations

This document has no IANA actions.

11. References

11.1. Normative References

- [Fetch] WHATWG, "Fetch", WHATWG Living Standard, <<https://fetch.spec.whatwg.org/>>. Commit snapshot: <<https://fetch.spec.whatwg.org/commit-snapshots/d9c7b6d31067a7eb13110950ef67db54c6362cfa/>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<https://www.rfc-editor.org/info/rfc6750>>.
- [RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <<https://www.rfc-editor.org/info/rfc7636>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <<https://www.rfc-editor.org/info/rfc8252>>.
- [RFC8707] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", RFC 8707, DOI 10.17487/RFC8707, February 2020, <<https://www.rfc-editor.org/info/rfc8707>>.
- [RFC9449] Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof of Possession (DPoP)", RFC 9449, DOI 10.17487/RFC9449, September 2023, <<https://www.rfc-editor.org/info/rfc9449>>.
- [RFC9700] Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "Best Current Practice for OAuth 2.0 Security", BCP 240, RFC 9700, DOI 10.17487/RFC9700, January 2025, <<https://www.rfc-editor.org/info/rfc9700>>.
- [W3C.service-workers] Chintala, M., Ed. and Y. Yanagisawa, Ed., "Service Workers Nightly", W3C Recommendation, 26 January 2026, <<https://www.w3.org/TR/2026/CRD-service-workers-20260126/>>. Latest version available at <<https://www.w3.org/TR/service-workers/>>.

[WebMessaging] WHATWG, "HTML - Cross-document messaging", WHATWG Living Standard, <<https://html.spec.whatwg.org/#web-messaging>>. Commit snapshot: <<https://html.spec.whatwg.org/commit-snapshots/b4b94b1db80aa55a143e1f74f6ad9d4a185f2333/#web-messaging>>.

11.2. Informative References

[COOKIES] van Kesteren, A. and J. Hofmann, "Cookies: HTTP State Management Mechanism", Work in Progress, Internet-Draft, draft-ietf-httpbis-layered-cookies-02, 21 May 2026, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-layered-cookies-02>>.

[CryptoKeyPair] Mozilla Developer Network, "CryptoKeyPair", <<https://developer.mozilla.org/en-US/docs/Web/API/CryptoKeyPair>>.

[HTML] WHATWG, "HTML", WHATWG Living Standard, <<https://html.spec.whatwg.org/>>. Commit snapshot: <<https://html.spec.whatwg.org/commit-snapshots/b4b94b1db80aa55a143e1f74f6ad9d4a185f2333/>>.

[OpenID] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 2", 15 December 2023, <https://openid.net/specs/openid-connect-core-1_0-errata2.html>.

[OWASPCheatSheet] "OWASP Cheat Sheet", <<https://cheatsheetseries.owasp.org/>>.

[RFC6265bis] Bingler, S., Ed., West, M., Ed., and J. Wilander, Ed., "Cookies: HTTP State Management Mechanism", Work in Progress, Internet-Draft, draft-ietf-httpbis-rfc6265bis-22, 1 December 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis-22>>.

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <<https://www.rfc-editor.org/info/rfc6819>>.

[SessionFixation] "Session Fixation", <https://owasp.org/www-community/attacks/Session_fixation>.

[Site] Mozilla Developer Network, "Site", <<https://developer.mozilla.org/en-US/docs/Glossary/Site>>.

[W3C.CSP3] West, M., Ed. and A. Sartori, Ed., "Content Security Policy Level 3", W3C Working Draft, 11 February 2026, <<https://www.w3.org/TR/2026/WD-CSP3-20260505/>>. Latest version available at <<https://www.w3.org/TR/CSP3/>>.

[W3C.IndexedDB] Becker, S., Ed., "Indexed Database API 3.0", W3C Working Draft, 13 August 2025, <<https://www.w3.org/TR/2025/WD-IndexedDB-3-20250813/>>. Latest version available at <<https://www.w3.org/TR/IndexedDB/>>.

- [W3C.SRI]** Braun, F., Ed., "Subresource Integrity", W3C Working Draft, 10 July 2025, <<https://www.w3.org/TR/2025/WD-sri-2-20250710/>>. Latest version available at <<https://www.w3.org/TR/sri-2/>>.
- [W3C.wasm-core-2]** Rossberg, A., Ed., "WebAssembly Core Specification", W3C Candidate Recommendation Draft, 10 February 2026, <<https://www.w3.org/TR/2026/CRD-wasm-core-2-20260210/>>. Latest version available at <<https://www.w3.org/TR/wasm-core-2/>>.
- [W3C.WebCryptoAPI]** Huigens, D., Ed., "Web Cryptography Level 2", W3C First Public Working Draft, 22 April 2025, <<https://www.w3.org/TR/2025/WD-webcrypto-2-20250422/>>. Latest version available at <<https://www.w3.org/TR/webcrypto-2/>>.
- [WebStorage]** WHATWG, "HTML - Web storage", WHATWG Living Standard, <<https://html.spec.whatwg.org/#webstorage>>. <<https://html.spec.whatwg.org/commit-snapshots/b4b94b1db80aa55a143e1f74f6ad9d4a185f2333/#webstorage>>.
- [WebWorker]** WHATWG, "HTML - Web workers", WHATWG Living Standard, <<https://html.spec.whatwg.org/#workers>>. Commit snapshot: <<https://html.spec.whatwg.org/commit-snapshots/b4b94b1db80aa55a143e1f74f6ad9d4a185f2333/#workers>>.

Acknowledgements

The authors would like to acknowledge the work of William Denniss and John Bradley, whose recommendations for native applications informed many of the best practices for browser-based applications. The authors would also like to thank Hannes Tschofenig and Torsten Lodderstedt, the attendees of the Internet Identity Workshop 27 session at which this BCP was originally proposed, and the following individuals who contributed ideas, feedback, and wording that shaped and formed the final specification:

Andy Barlow, Andy Newton, Annabelle Backman, Brian Campbell, Brock Allen, Christian Mainka, Damien Bowden, Daniel Fett, Deb Cooley, Elar Lang, Emmanuel Gautier, Erik Kline, Éric Vyncke, Eva Sarafianou, Filip Skokan, George Fletcher, Hannes Tschofenig, Janak Amarasena, John Bradley, Joseph Heenan, Justin Richer, Karl McGuinness, Karsten Meyer zu Selhausen, Leo Tohill, Louis Jannett, Marc Blanchet, Martin Thomson, Matthew Bocci, Mike Bishop, Mike Jones, Mohamed Boucadair, Orie Steele, Qin Wu, Rifaat Shekh-Yusef, Roman Danyliw, Sean Kelleher, Thomas Broyer, Thomas Fossati, Tomek Stojek, Torsten Lodderstedt, Vittorio Bertocci, Watson Ladd, William Duncan, and Yannick Majoros.

Authors' Addresses

Aaron Parecki

Okta

Email: aaron@parecki.com

URI: <https://aaronparecki.com>

Philippe De Ryck

Pragmatic Web Security

Email: philippe@pragmaticwebsecurity.com**David Waite**

Ping Identity

Email: dwaite@pingidentity.com