Workgroup: Web Authorization Protocol
Internet-Draft:
draft-ietf-oauth-browser-based-apps-13
Published: 13 March 2023
Intended Status: Best Current Practice
Expires: 14 September 2023
Authors: A. Parecki
Okta
D. Waite
Ping Identity

OAuth 2.0 for Browser-Based Apps

Abstract

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

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the Web Authorization Protocol Working Group mailing list (oauth@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/oauth/.

Source for this draft and an issue tracker can be found at https://github.com/oauth-wg/oauth-browser-based-apps.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 14 September 2023.

Copyright Notice

Table of Contents

1. Introduction

This specification describes the current best practices for implementing OAuth 2.0 authorization flows in applications executing in a browser.

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. This document is formally known as [RFC8252] or BCP 212, but nicknamed "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 in native applications, including incorporating additional OAuth extensions where needed.

OAuth 2.0 for Browser-Based Apps addresses the similarities between implementing OAuth for native apps and browser-based apps, and includes additional considerations when apps are running in a browser. This is primarily focused on OAuth, except where OpenID Connect provides additional considerations.

Many of these recommendations are derived from the OAuth 2.0 Security Best Current Practice [oauth-security-topics] and browser-based apps are expected to follow those recommendations as well. This draft expands on and further restricts various recommendations in [oauth-security-topics].

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 [RFC2119].

3. Terminology

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] and [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".

While this document often refers to "JavaScript apps", this is not intended to be exclusive to JavaScript. The recommendations and considerations herein also apply to other languages that execute code in the browser, such as Web Assembly.

4. Overview

At the time that OAuth 2.0 [RFC6749] and [RFC6750] were created, 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 flow which would require a cross-origin POST request. This was one of the motivations for the definition of the Implicit flow, 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 flow, generally involving vulnerabilities associated with the exposure of the access token in the URL. See Section 10.9 for an analysis of these attacks and the drawbacks of using the Implicit flow in browsers. Additional attacks and security considerations can be found in [oauth-security-topics].

In recent years, widespread adoption of Cross-Origin Resource Sharing (CORS), which enables exceptions to the same-origin policy, allows browser-based apps 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. In this flow, the access token is never exposed in the less-secure front channel. Furthermore, adding PKCE 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 flow with PKCE. There are various architectural patterns for deploying browser-based apps, both with and without a corresponding server-side component, each with their own trade-offs and considerations, discussed further in this document. Additional considerations apply for first-party common-domain apps.

In summary, browser-based applications using the Authorization Code flow:

- MUST use PKCE ([RFC7636]) when obtaining an access token (Section 7.1)
- MUST Protect themselves against CSRF attacks (Section 7.3) by either:
  - ensuring the authorization server supports PKCE, or
  - by using the OAuth 2.0 "state" parameter or the OpenID Connect "nonce" parameter to carry one-time use CSRF tokens
- MUST Register one or more redirect URIs, and use only exact registered redirect URIs in authorization requests (Section 7.2)

In summary, OAuth 2.0 authorization servers supporting browser-based applications using the Authorization Code flow:

- MUST Require exact matching of registered redirect URIs (Section 7.2)
- MUST Support the PKCE extension (Section 7.1)
- MUST NOT issue access tokens in the authorization response (Section 10.9)
- If issuing refresh tokens to browser-based applications (Section 8), then:
  - MUST rotate refresh tokens on each use or use sender-constrained refresh tokens, and
  - MUST set a maximum lifetime on refresh tokens or expire if they are not used in some amount of time
  - when issuing a rotated refresh token, MUST NOT extend the lifetime of the new refresh token beyond the lifetime of the original refresh token if the refresh token has a preestablished expiration time

5. First-Party Applications

While OAuth was initially created to allow third-party applications to access an API on behalf of a user, it has proven to be useful in a first-party scenario as well. First-party apps are applications where the same organization provides both the API and the application.

Examples of first-party applications are a web email client provided by the operator of the email account, or a mobile banking application created by bank itself. (Note that there is no requirement that the application actually be developed by the same company; a mobile banking application developed by a contractor that is branded as the bank's application is still considered a first-party application.) The first-party app consideration is about the user's relationship to the application and the service.

To conform to this best practice, first-party browser-based applications using OAuth or OpenID Connect MUST use a redirect-based flow (such as the OAuth Authorization Code flow) as described later in this document.

The resource owner password credentials grant MUST NOT be used, as described in [oauth-security-topics] Section 2.4. Instead, by using the Authorization Code flow and redirecting the user to the authorization server, this provides the authorization server the opportunity to prompt the user for multi-factor 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 code.

6. Application Architecture Patterns

Here are the main architectural patterns available when building browser-based applications.

*single-domain, not using OAuth

*a JavaScript application with a stateful backend component

  -storing tokens and proxying all requests (BFF Proxy)

  -obtaining tokens and passing them to the frontend (Token-Mediating Backend)

*a JavaScript application obtaining access tokens

  -via code executed in a browsing context

-through a Service Worker

These architectures have different use cases and considerations.

6.1. Single-Domain Browser-Based Apps (not using OAuth)

For simple system architectures, such as when the JavaScript application is served from a domain that can share cookies with the domain of the API (resource server) and the authorization server, OAuth adds additional attack vectors that could be avoided with a different solution.

In particular, using any redirect-based mechanism of obtaining an access token enables the redirect-based attacks described in [oauth-security-topics] Section 4, but if the application, authorization server and resource server share a domain, then it is unnecessary to use a redirect mechanism to communicate between them.

An additional concern with handling access tokens in a browser is that in case of successful cross-site scripting (XSS) attack, tokens could be read and further used or transmitted by the injected code if no secure storage mechanism is in place.

As such, it could be considered to use an HTTP-only cookie between the JavaScript application and API so that the JavaScript code can't access the cookie value itself. The Secure cookie attribute should be used to ensure the cookie is not included in unencrypted HTTP requests. Additionally, the SameSite cookie attribute can be used to counter some CSRF attacks, but should not be considered the extent of the CSRF protection, as described in [draft-ietf-httpbis-rfc6265bis]

OAuth was originally created for third-party or federated access to APIs, so it may not be the best solution in a common-domain deployment. That said, there are still some advantages in using OAuth even in a common-domain architecture:

* Allows more flexibility in the future, such as if you were to later add a new domain to the system. With OAuth already in place, adding a new domain wouldn't require any additional rearchitecting.

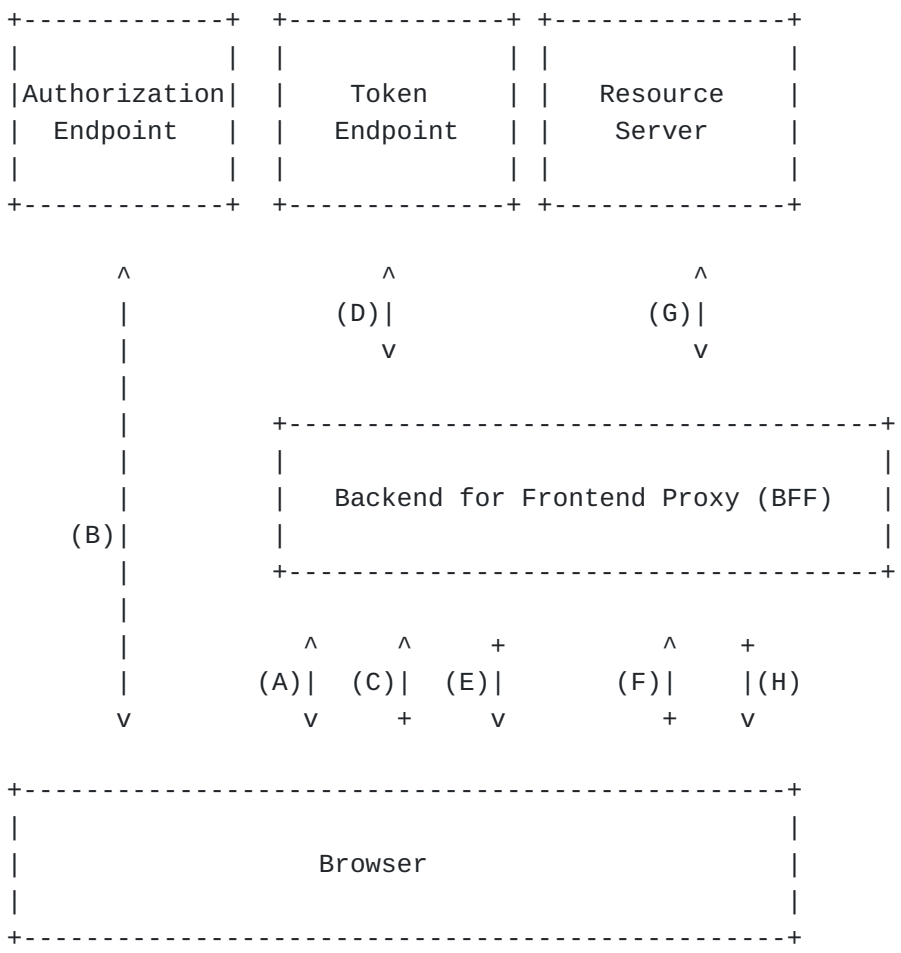
* Being able to take advantage of existing library support rather than writing bespoke code for the integration.

* Centralizing login and multifactor authentication support, account management, and recovery at the OAuth server, rather than making it part of the application logic.

*Splitting of responsibilities between authenticating a user and serving resources

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

6.2. Backend For Frontend (BFF) Proxy



In this architecture, commonly referred to as "backend for frontend" or "BFF", the JavaScript code is loaded from a BFF Proxy server (A) that has the ability to execute code and handle the full OAuth flow itself. This enables the ability to keep the request to obtain an access token outside the JavaScript application.

Note that this BFF Proxy is not the Resource Server, it is the OAuth client and would be later accessing data at a separate resource server after obtaining tokens.

In this case, the BFF Proxy initiates the OAuth flow itself, by redirecting the browser to the authorization endpoint (B). When the user is redirected back, the browser delivers the authorization code to the BFF Proxy (C), where it can then exchange it for an access

token at the token endpoint (D) using its client secret and PKCE code verifier. The BFF Proxy then keeps the access token and refresh token stored internally, and creates a separate session with the browser-based app via a traditional browser cookie (E).

When the JavaScript application in the browser wants to make a request to the Resource Server, it instead makes the request to the BFF Proxy (F), and the BFF Proxy will make the request with the access token to the Resource Server (G), and forward the response (H) back to the browser.

(Common examples of this architecture are an Angular front-end with a .NET backend, or a React front-end with a Spring Boot backend.)

The BFF Proxy SHOULD be considered a confidential client, and issued its own client secret. The BFF Proxy SHOULD use the OAuth 2.0 Authorization Code grant with PKCE to initiate a request for an access token. Detailed recommendations for confidential clients can be found in [oauth-security-topics] Section 2.1.1.

In this scenario, the connection between the browser and BFF Proxy SHOULD be a session cookie provided by the BFF Proxy.

While the security of this model is strong, since the OAuth tokens are never sent to the browser, there are performance and scalability implications of deploying a BFF proxy server and routing all JS requests through the server. If routing every API request through the BFF proxy is prohibitive, you may wish to consider one of the alternative architectures below.

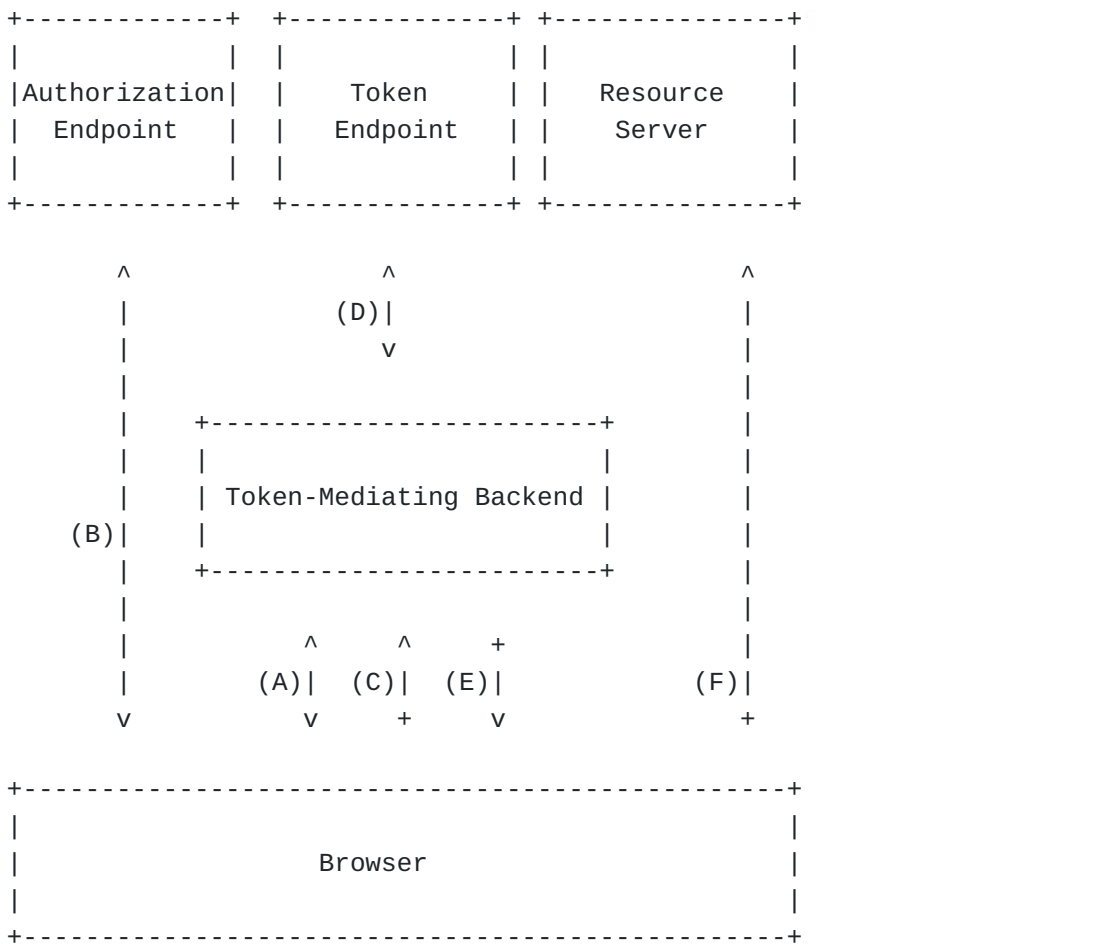
6.2.1. Security considerations

Security of the connection between code running in the browser and this BFF Proxy is assumed to utilize browser-level protection mechanisms. Details are out of scope of this document, but many recommendations can be found in the OWASP Cheat Sheet series (https://cheatsheetseries.owasp.org), such as setting an HTTP-only and Secure cookie to authenticate the session between the browser and BFF Proxy. Additionally, cookies MUST be protected from leakage by other means, such as logs.

In this architecture, tokens are never sent to the front-end and are never accessible by any JavaScript code, so it fully protects against XSS attackers stealing tokens. However, an XSS attacker may still be able to make authenticated requests to the BFF Proxy which will in turn make requests to the resource server including the user's legitimate token. While the attacker is unable to extract and use the access token elsewhere, they could still effectively make authenticated requests to the resource server.

6.3. Token-Mediating Backend

An alternative to a full BFF where all resource requests go through the backend is to use a token-mediating backend which obtains the tokens and then forwards the tokens to the browser.



The frontend code makes a request to the Token-Mediating Backend (A), and the backend initiates the OAuth flow itself, by redirecting the browser to the authorization endpoint (B). When the user is redirected back, the browser delivers the authorization code to the application server (C), where it can then exchange it for an access token at the token endpoint (D) using its client secret and PKCE code verifier. The backend delivers the tokens to the browser (E), which stores them for later use. The browser makes requests to the resource server directly (F) including the token it has stored.

The main advantage this architecture provides over the full BFF architecture previously described is that the backend service is only involved in the acquisition of tokens, and doesn't have to proxy every request in the future. Routing every API call through a backend can be expensive in terms of performance and latency, and can create challenges in deploying the application across many

regions. Instead, routing only the token acquisition through a backend means fewer requests are made to the backend. This improves the performance and reduces the latency of requests from the frontend, and reduces the amount of infrastructure needed in the backend.

Similar to the previously described BFF Proxy pattern, The Token-Mediating Backend SHOULD be considered a confidential client, and issued its own client secret. The Token-Mediating Backend SHOULD use the OAuth 2.0 Authorization Code grant with PKCE to initiate a request for an access token. Detailed recommendations for confidential clients can be found in [oauth-security-topics] Section 2.1.1.

In this scenario, the connection between the browser and Token-Mediating Backend SHOULD be a session cookie provided by the backend.

The Token-Mediating Backend SHOULD cache tokens it obtains from the authorization server such that when the frontend needs to obtain new tokens, it can do so without the additional round trip to the authorization server if the tokens are still valid.

The frontend SHOULD NOT persist tokens in local storage or similar mechanisms; instead, the frontend SHOULD store tokens only in memory, and make a new request to the backend if no tokens exist. This provides fewer attack vectors for token exfiltration should an XSS attack be successful.

Editor's Note: A method of implementing this architecture is described by the [tmi-bff] draft, although it is currently an expired individual draft and has not been proposed for adoption to the OAuth Working Group.

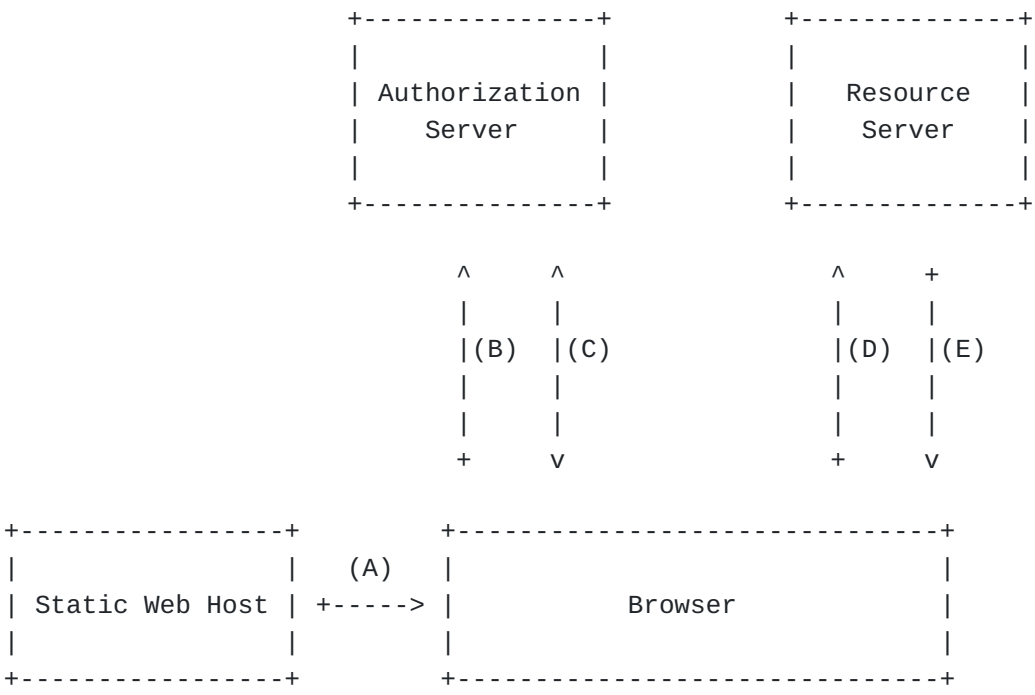
6.3.1. Security Considerations

If the backend caches tokens from the authorization server, it presents scope elevation risks if applied indiscriminately. If the token cached by the authorization server features a superset of the scopes requested by the frontend, the backend SHOULD NOT return it to the frontend; instead it SHOULD perform a new request with the smaller set of scopes to the authorization server.

In the case of a successful XSS attack, the attacker may be able to access the tokens if the tokens are persisted in the frontend, but is less likely to be able to access the tokens if they are stored only in memory. However, a successful XSS attack will also allow the attacker to call the Token-Mediating Backend itself to retrieve the cached token or start a new OAuth flow.

6.4. JavaScript Applications obtaining tokens directly

This section describes the architecture of a JavaScript application obtaining tokens from the authorization server itself, with no intermediate proxy server and no backend component.



In this architecture, the JavaScript code is first loaded from a static web host into the browser (A), and the application then runs in the browser. This application is considered a public client, since there is no way to issue it a client secret in this model.

The code in the browser initiates the Authorization Code flow with the PKCE extension (described in Section 7) (B) above, and obtains an access token via a POST request (C).

The application is then responsible for storing the access token (and optional refresh token) as securely as possible using appropriate browser APIs, described in Section 9.

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

In this scenario, the Authorization Server and Resource Server MUST support the necessary CORS headers to enable the JavaScript code to make these POST requests from the domain on which the script is executing. (See Section 10.7 for additional details.)

Besides the general risks of XSS, if tokens are stored or handled by the browser, XSS poses an additional risk of token exfiltration. In this architecture, the JavaScript application is storing the access token so that it can make requests directly to the resource server. There are two primary methods by which the application can acquire tokens, with different security considerations of each.

6.4.1. Acquiring tokens from the Browsing Context

If the JavaScript executing in the browsing context will be making requests directly to the resource server, the simplest mechanism is to acquire and store the tokens somewhere accessible to the JavaScript code. This will typically involve JavaScript code initiating the Authorization Code flow and exchanging the authorization code for an access token, and then storing the access token obtained. There are a number of different options for storing tokens, each with different tradeoffs, described in Section 9.

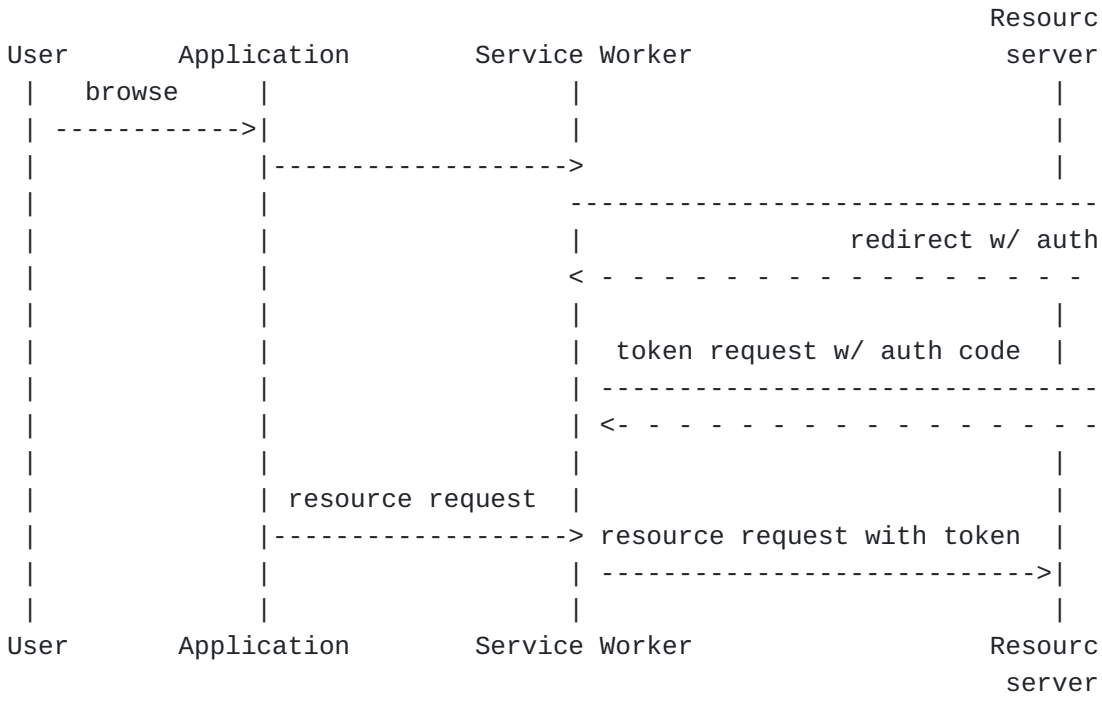
This method poses a particular risk in the case of a successful XSS attack. In case of a successful XSS attack, the injected code will have full access to the stored tokens and can exfiltrate them to the attacker.

6.4.2. Acquiring tokens from a Service Worker

In this model, a Service Worker is responsible for obtaining tokens from the authorization server and making requests to the resource server.

Service workers are run in a separate context from the DOM, have no access to the DOM, and the DOM has no access to the service worker or the storage available to the service worker. This makes service workers the most secure place to acquire and store tokens, as an XSS attack would be unable to exfiltrate the tokens.

In this architecture, a service worker intercepts calls from the frontend to the resource server. As such, it completely isolates calls to the authorization server from XSS attack surface, as all tokens are safely kept in the service worker context without any access from other JavaScript contexts. The service worker is then solely responsible for adding the token in the authorization header to calls to the resource server.



6.4.2.1. Implementation Guidelines

- The service worker MUST initiate the OAuth 2.0 Authorization Code grant with PKCE itself.
- The service worker MUST intercept the authorization code when the *authorization server* redirects to the application.
- The service worker implementation MUST then initiate the token request itself.
- The service worker MUST not transmit tokens, authorization codes or PKCE secrets (e.g. code verifier) to the frontend application.
- The service worker MUST block authorization requests and token requests initiating from the frontend application in order to avoid any front-end side-channel for getting tokens: the only way of starting the authorization flow should be through the service worker. This protects against re-authorization from XSS-injected code.
- The application MUST register the Service Worker before running any code interacting with the user.

See Section 9.2 for details on storing tokens from the Service Worker.

6.4.2.2. Security Considerations

A successful XSS attack on an application using this Service Worker pattern would be unable to exfiltrate existing tokens stored by the application. However, an XSS attacker may still be able to cause the Service Worker to make authenticated requests to the resource server including the user's legitimate token.

In case of a vulnerability leading to the Service Worker not being registered, an XSS attack would result in the attacker being able to initiate a new OAuth flow to obtain new tokens itself.

To prevent the Service Worker from being unregistered, the Service Worker registration MUST happen as first step of the application start, and before any user interaction. Starting the Service worker before the rest of the application, and the fact that there is no way to remove a Service Worker from an active application, reduces the risk of an XSS attack being able to prevent the Service Worker from being registered.

7. Authorization Code Flow

Browser-based applications that are public clients and use the Authorization Code grant type described in Section 4.1 of OAuth 2.0 [RFC6749] MUST also follow these additional requirements described in this section.

7.1. Initiating the Authorization Request from a Browser-Based Application

Browser-based applications that are public clients MUST implement the Proof Key for Code Exchange (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 the client instance that exchanges the authorization code is the same one that initiated the flow.

7.2. 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 [oauth-security-topics] Section 4.1.1. this helps to prevent attacks targeting the authorization code.

7.3. 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:

- using PKCE, and confirming that the authorization server supports PKCE
- using a unique value for the OAuth 2.0 "state" parameter to carry a CSRF token
- if the application is using OpenID Connect, by using and verifying the OpenID Connect "nonce" parameter as described in [OpenID]

See Section 2.1 of [oauth-security-topics] for additional details.

8. Refresh Tokens

Refresh tokens provide a way for applications to obtain a new access token when the initial access token expires. With public clients, the risk of a leaked refresh token is greater than leaked access tokens, since an attacker may be able to continue using the stolen refresh token to obtain new access tokens potentially without being detectable by the authorization server.

Javascript-accessible storage mechanisms like *Local Storage* provide an attacker with several opportunities by which a refresh token can be leaked, just as with access tokens. As such, these mechanisms are considered a higher risk for handling refresh tokens.

Authorization servers may choose whether or not to issue refresh tokens to browser-based applications. [oauth-security-topics] 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 [oauth-security-topics] around 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 [oauth-security-topics] Section 4.13.2
- MUST either set a maximum lifetime on refresh tokens OR expire if the refresh token has not been used within some amount of time
- upon issuing a rotated refresh token, MUST NOT 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

For example:

* A user authorizes an application, issuing an access token that lasts 1 hour, and a refresh token that lasts 24 hours
* After 1 hour, 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 1 hour, and a new refresh token that lasts 23 hours
* This continues until 24 hours pass from the initial authorization
* At this point, when the application attempts to use the refresh token after 24 hours, the request will fail and the application will have to involve the user in a new authorization request

By limiting the overall refresh token lifetime to the lifetime of the initial refresh token, this ensures a stolen refresh token cannot be used indefinitely.

Authorization servers MAY set different policies around refresh token issuance, lifetime and expiration for browser-based applications compared to other public clients.

9. Token Storage in the Browser

When using an architectural pattern that involves the browser-based code obtaining tokens itself, the application will ultimately need to store the tokens it acquires for later use. This applies to both the Token-Mediating Backend architecture as well as any architecture where the JavaScript code is the OAuth client itself and does not have a corresponding backend component.

This section is primarily concerned with the ability for an attacker to exfiltrate the tokens from where they are stored. Token exfiltration may occur via an XSS attack, via injected code from a browser extension, via malicious code deployed to the application such as via upstream dependencies of a package management system, or by the attacker getting access to the filesystem of the user's machine via malware.

There are a number of storage options available to browser-based applications, and more may be created in the future. The different

options have different use cases and considerations, and there is no clear "best" option that applies to every scenario. Tokens can be:

* Stored and managed by a Service Worker

* Stored in memory only, in particular stored in a closure variable rather than an object property

* Stored in LocalStorage, SessionStorage, or IndexedDB

* Stored in an encrypted format using the WebCrypto API to encrypt and decrypt from storage

9.1. Cookies

The JavaScript Cookie API is a mechanism that is technically possible to use as storage from JavaScript, but is NOT RECOMMENDED as a place to store tokens that will be later accessed from JavaScript. (Note that this statement does not affect the BFF pattern described in Section 6.2 since in that pattern the tokens are never accessible to the browser-based code.)

When JavaScript code stores a token, the intent is for it to be able to retrieve the token for later use in an API call. Using the Cookie API to store the token has the unintended side effect of the browser also sending the token to the web server the next time the app is loaded, or on any API calls the app makes to its own backend.

Illustrating this example with the diagram in Section 6.4, the app would acquire the tokens in step C, store them in a cookie, and the next time the app loads from the Static Web Host, the browser would transmit the tokens in the Cookie header to the Static Web Host unnecessarily. Instead, the tokens should be stored using an API that is only accessible to JavaScript, such as the methods described below, so that the tokens are only sent outside the browser when intended.

9.2. Token Storage in a Service Worker

Obtaining and storing the tokens with a service worker is the most secure option for unencrypted storage, as that isolates the tokens from XSS attacks, as described in Section 6.4.2.

The Service Worker MUST NOT store tokens in any persistent storage API that is shared with the main window. For example, the IndexedDB storage is shared between the browsing context and Service Worker, so is not a suitable place for the Service Worker to persist data that should remain inaccessible to the main window.

Service Workers are not guaranteed to run persistently, and may be shut down by the browser for various reasons. This should be taken into consideration when implementing this pattern, until a persistent storage API that is isolated to Service Workers is available in browsers.

This, like the other unencrypted options, do not provide any protection against exfiltration from the filesystem.

9.3. In-Memory Token Storage

If using a service worker is not a viable option, the next most secure option is to store tokens in memory only. To prevent XSS attackers from exfiltrating the tokens, a "token manager" class can store the token in a closure variable (rather than an object property), and manage all calls to the resource server itself, never letting the access token be accessible outside this manager class.

However, the major downside to this approach is that the tokens will not be persisted between page reloads. If that is a property you would like, then the next best options are one of the persistent browser storage APIs.

9.4. Persistent Token Storage

The persistent storage APIs currently available as of this writing are LocalStorage, SessionStorage, and IndexedDB.

LocalStorage persists between page reloads as well as is shared across all tabs. This storage is accessible to the entire origin, and persists longer term. LocalStorage does not protect against XSS attacks, as the attacker would be running code within the same origin, and as such, would be able to read the contents of the LocalStorage.

SessionStorage is similar to LocalStorage, except that SessionStorage is cleared when a browser tab is closed, and is not shared between multiple tabs open to pages on the same origin. This slightly reduces the chance of a successful XSS attack, since a user who clicks a link carrying an XSS payload would open a new tab, and wouldn't have access to the existing tokens stored. However there are still other variations of XSS attacks that can compromise this storage.

IndexedDB is a persistent storage mechanism like LocalStorage, but is shared between multiple tabs as well as between the browsing context and Service Workers. For this reason, IndexedDB SHOULD NOT be used by a Service Worker if attempting to use the Service Worker to isolate the front-end from XSS attacks.

9.5. Filesystem Considerations for Browser Storage APIs

In all cases, as of this writing, browsers ultimately store data in plain text on the filesystem. Even if an application does not suffer from an XSS attack, other software on the computer may be able to read the filesystem and exfiltrate tokens from the storage.

The [WebCrypto] API provides a mechanism for JavaScript code to generate a private 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 WebCrypto specification only ensures that the key is not exportable to the browser code, but 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.

9.6. Sender-Constrained Tokens

Sender-constrained tokens require that the OAuth client 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 need to also steal the private key.

One method of implementing sender-constrained tokens in a way that is usable from browser-based apps is [DPoP].

Using sender-constrained tokens shifts the challenge of securely storing the token to securely storing the private key.

If an application is using sender-constrained tokens, the secure storage of the private key is more important than the secure storage of the token. Ideally the application should use a non-exportable private key, such as generating one with the [WebCrypto] API. With an unencrypted token in LocalStorage 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 XSS attacks, such as by generating and storing it in a closure variable or in a Service Worker. This is similar to the

considerations for storing tokens in a Service Worker, as described in Section 9.2.

10. Security Considerations

10.1. Cross-Site Scripting Attacks (XSS)

For all known architectures, all precautions MUST be taken to prevent cross-site scripting (XSS) attacks. In general, XSS attacks are a huge risk, and can lead to full compromise of the application.

If tokens are handled or accessible by the browser, there is a risk that a XSS attack can lead to token exfiltration.

Even if tokens are never sent to the frontend and are never accessible by any JavaScript code, an XSS attacker may still be able to make authenticated requests to the resource server by mimicking legitimate code in the browsing context. For example, the attacker may make a request to the BFF Proxy which will in turn make requests to the resource server including the user's legitimate token. In the Service Worker example, the attacker may make an API call to the resource server, and the Service Worker will intercept the request and add the access token to the request. While the attacker is unable to extract and use the access token elsewhere, they can still effectively make authenticated requests to the resource server to steal or modify data.

10.2. Reducing the Impact of Token Exfiltration

If tokens are ever accessible to the browser or to any JavaScript code, there is always a risk of token exfiltration. The particular risk may change depending on the architecture chosen. Regardless of the particular architecture chosen, these additional security considerations limit the impact of token exfiltration:

* The authorization server SHOULD restrict access tokens to strictly needed resources, to avoid escalating the scope of the attack.

* To avoid information disclosure from ID Tokens, the authorization server SHOULD NOT include any ID token claims that aren't used by the frontend.

* Refresh tokens should be used in accordance with the guidance in Section 8.

10.3. Registration of Browser-Based Apps

Browser-based applications (with no backend) are considered public clients as defined by Section 2.1 of OAuth 2.0 [RFC6749], and MUST

be registered with the authorization server as such. Authorization servers MUST record the client type in the client registration details in order to identify and process requests accordingly.

Authorization servers MUST require that browser-based applications register one or more redirect URIs.

10.4. Client Authentication

Since a browser-based application's source code is delivered to the end-user's browser, it cannot contain provisioned secrets. As such, a browser-based app with native OAuth support is considered a public client as defined by Section 2.1 of OAuth 2.0 [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], it is NOT RECOMMENDED for authorization servers to require client authentication of browser-based applications using a shared secret, as this serves little 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 10.5 below).

10.5. Client Impersonation

As stated in Section 10.2 of OAuth 2.0 [RFC6749], the authorization server SHOULD NOT process authorization requests automatically without user consent or interaction, except when the identity of the client can be assured.

If authorization servers restrict redirect URIs to a fixed set of absolute HTTPS URIs, preventing 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.

10.6. Authorization Server Mix-Up Mitigation

Authorization server mix-up attacks mark a severe threat to every client that supports at least two authorization servers. To conform

to this BCP such clients MUST apply countermeasures to defend against mix-up attacks.

It is RECOMMENDED to defend against mix-up attacks by identifying and validating the issuer of the authorization response. This can be achieved either by using the "iss" response parameter, as defined in [oauth-iss-auth-resp], or by using the "iss" Claim of the ID token when OpenID Connect is used.

Alternative countermeasures, such as using distinct redirect URIs for each issuer, SHOULD only be used if identifying the issuer as described is not possible.

Section 4.4 of [oauth-security-topics] provides additional details about mix-up attacks and the countermeasures mentioned above.

10.7. Cross-Domain Requests

To complete the Authorization Code flow, the browser-based application will need to exchange the authorization code for an access token at the token endpoint. If the authorization server provides additional endpoints to the application, such as metadata URLs, dynamic client registration, revocation, introspection, discovery or user info endpoints, these endpoints may also be accessed by the browser-based app. Since these requests will be made from a browser, authorization servers MUST support the necessary CORS headers (defined in [Fetch]) to allow the browser to make the request.

This specification does not include guidelines for deciding whether a CORS policy for the token endpoint should be a wildcard origin or more restrictive. Note, however, that the browser will attempt to GET or POST to the API endpoint before knowing any CORS policy; it simply hides the succeeding or failing result from JavaScript if the policy does not allow sharing.

10.8. Content Security Policy

A browser-based application that wishes to use either long-lived refresh tokens or privileged scopes SHOULD restrict its JavaScript execution to a set of statically hosted scripts via a Content Security Policy ([CSP3]) or similar mechanism. A strong Content Security Policy can limit the potential attack vectors for malicious JavaScript to be executed on the page.

10.9. OAuth Implicit Flow

The OAuth 2.0 Implicit flow (defined in Section 4.2 of OAuth 2.0 [RFC6749]) works by the authorization server issuing an access token in the authorization response (front channel) without the 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.

Authorization servers MUST NOT issue access tokens in the authorization response, and MUST issue access tokens only from the token endpoint.

10.9.1. Attacks on the Implicit Flow

Many attacks on the Implicit flow described by [RFC6819] and Section 4.1.2 of [oauth-security-topics] do not have sufficient mitigation strategies. The following sections describe the specific attacks that cannot be mitigated while continuing to use the Implicit flow.

10.9.1.1. Threat: 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 directly get access to the authorization response including the access token. Several methods of performing this attack are described in detail in [oauth-security-topics].

10.9.1.2. Threat: Access Token Leak in Browser History

An attacker could obtain the access token from the browser's history. The countermeasures recommended by [RFC6819] are limited to using short expiration times for tokens, and indicating 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 [oauth-security-topics].

10.9.1.3. Threat: 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 man-in-the-middle attacks are typically out of scope of basic security recommendations to prevent, in the case of browser-based apps 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 that 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.

10.9.1.4. Threat: Access Token Leak to Third-Party Scripts

It is relatively common to use third-party scripts in browser-based apps, such as analytics tools, crash reporting, and even things like a Facebook or Twitter "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.

10.9.2. Countermeasures

In addition to the countermeasures described by [RFC6819] and [oauth-security-topics], using the Authorization Code flow with PKCE extension prevents the attacks described above by avoiding returning the access token in the redirect response.

When PKCE is used, if an authorization code is stolen in transport, the attacker is unable to do anything with the authorization code.

10.9.3. Disadvantages of the Implicit Flow

There are several additional reasons the Implicit flow is disadvantageous compared to using the standard Authorization Code flow.

* 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 flow requires additional code, more upkeep and understanding of the related security considerations, while limiting the authorization server to just the Authorization Code flow reduces the attack surface of the implementation.

*If the JavaScript application gets wrapped into a native app, then [RFC8252] also requires the use of the Authorization Code flow with PKCE anyway.

In OpenID Connect, the ID Token is sent in a known format (as a JWT), and digitally signed. Returning an ID token using the Implicit flow (response_type=id_token) requires the client validate the JWT signature, as malicious parties could otherwise craft and supply fraudulent ID tokens. Performing OpenID Connect using the Authorization Code flow provides the benefit of the client not needing to verify the JWT signature, as the ID token will have been fetched over an HTTPS connection directly from the authorization server. Additionally, in many cases an application will request both an ID token and an access token, so it is simplier and provides fewer attack vectors to obtain both via the Authorization Code flow.

10.9.4. Historic Note

Historically, the Implicit flow provided an advantage to browser-based apps 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 Cross Origin Resource Sharing (CORS) was widespread in browsers, the Implicit flow 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 "Session history and navigation" 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 apps for many purposes. This means modern browser-based apps can use the unmodified OAuth 2.0 Authorization Code flow, 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.

10.10. Additional Security Considerations

The OWASP Foundation (https://www.owasp.org/) maintains a set of security recommendations and best practices for web applications, and it is RECOMMENDED to follow these best practices when creating an OAuth 2.0 Browser-Based application.

11. IANA Considerations

This document does not require any IANA actions.

12. References

12.1. Normative References

[CSP3] West, M., "Content Security Policy", October 2018, <https://www.w3.org/TR/CSP3/>.

[draft-ietf-httpbis-rfc6265bis] Chen, L., Englehardt, S., West, M., and J. Wilander, "Cookies: HTTP State Management Mechanism", October 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis>.

[Fetch] whatwg, "Fetch", 2018, <https://fetch.spec.whatwg.org/>.

[oauth-iss-auth-resp] Meyer zu Selhausen, K. and D. Fett, "OAuth 2.0 Authorization Server Issuer Identifier in Authorization Response", January 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-iss-auth-resp>.

[oauth-security-topics] Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", April 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics>.

[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>.

[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>.

[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>.

[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>.

12.2. Informative References

[DPoP] Fett, D., Cambpell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "Demonstrating Proof-of-Possession at the Application Layer", n.d., <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop>.

[HTML] whatwg, "HTML", 2020, <https://html.spec.whatwg.org/>.

[OpenID] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect", November 2014, <https://openid.net/specs/openid-connect-core-1_0.html>.

[tmi-bff] Bertocci, V. and B. Cambpell, "Token Mediating and session Information Backend For Frontend", April 2021, <https://datatracker.ietf.org/doc/html/draft-bertocci-oauth2-tmi-bff-01>.

[WebCrypto] Huigens, D., "Web Cryptography API", November 2022, <https://w3c.github.io/webcrypto/>.

Appendix A. Server Support Checklist

OAuth authorization servers that support browser-based apps MUST:

1. Support PKCE [RFC7636]. Required to protect authorization code grants sent to public clients. See Section 7.1

2. NOT support the Resource Owner Password grant for browser-based clients.

3. NOT support the Implicit grant for browser-based clients.

4. Require "https" scheme redirect URIs.

5. Require exact matching of registered redirect URIs.

6. Support cross-domain requests at the token endpoint in order to allow browsers to make the authorization code exchange request. See Section 10.7

7. Not assume that browser-based clients can keep a secret, and SHOULD NOT issue secrets to applications of this type.

8. Follow the [oauth-security-topics] recommendations on refresh tokens, as well as the additional requirements described in Section 8.

Appendix B. Document History

[[To be removed from the final specification]]

-13

*Corrected some uses of "DOM"

*Consolidated CSRF recommendations into normative part of the document

*Added links from the summary into the later sections

*Described limitations of Service Worker storage

*Minor editorial improvements

-12

*Revised overview and server support checklist to bring them up to date with the rest of the draft

*Added a new section about options for storing tokens

*Added a section on sender-constrained tokens and a reference to DPoP

*Rephrased the architecture patterns to focus on token acquisition

*Added a section discussing why not to use the Cookie API to store tokens

-11

*Added a new architecture pattern: Token-Mediating Backend

*Revised and added clarifications for the Service Worker pattern

*Editorial improvements in descriptions of the different architectures

*Rephrased headers

-10

*Revised the names of the architectural patterns

*Added a new pattern using a service worker as the OAuth client to manage tokens

*Added some considerations when storing tokens in Local or Session Storage

-09

*Provide additional context for the same-domain architecture pattern

*Added reference to draft-ietf-httpbis-rfc6265bis to clarify that SameSite is not the only CSRF protection measure needed

*Editorial improvements

-08

*Added a note to use the "Secure" cookie attribute in addition to SameSite etc

*Updates to bring this draft in sync with the latest Security BCP

*Updated text for mix-up countermeasures to reference the new "iss" extension

*Changed "SHOULD" for refresh token rotation to MUST either use rotation or sender-constraining to match the Security BCP

*Fixed references to other specs and extensions

*Editorial improvements in descriptions of the different architectures

-07

*Clarify PKCE requirements apply only to issuing access tokens

*Change "MUST" to "SHOULD" for refresh token rotation

*Editorial clarifications

-06

*Added refresh token requirements to AS summary

*Editorial clarifications

-05

*Incorporated editorial and substantive feedback from Mike Jones

*Added references to "nonce" as another way to prevent CSRF attacks

*Updated headers in the Implicit Flow section to better represent the relationship between the paragraphs

-04

*Disallow the use of the Password Grant

*Add PKCE support to summary list for authorization server requirements

*Rewrote refresh token section to allow refresh tokens if they are time-limited, rotated on each use, and requiring that the rotated refresh token lifetimes do not extend past the lifetime of the initial refresh token, and to bring it in line with the Security BCP

*Updated recommendations on using state to reflect the Security BCP

*Updated server support checklist to reflect latest changes

*Updated the same-domain JS architecture section to emphasize the architecture rather than domain

*Editorial clarifications in the section that talks about OpenID Connect ID tokens

-03

*Updated the historic note about the fragment URL clarifying that the Session History API means browsers can use the unmodified authorization code flow

*Rephrased "Authorization Code Flow" intro paragraph to better lead into the next two sections

*Softened "is likely a better decision to avoid using OAuth entirely" to "it may be..." for common-domain deployments

*Updated abstract to not be limited to public clients, since the later sections talk about confidential clients

*Removed references to avoiding OpenID Connect for same-domain architectures

*Updated headers to better describe architectures (Apps Served from a Static Web Server -> JavaScript Applications without a Backend)

*Expanded "same-domain architecture" section to better explain the problems that OAuth has in this scenario

*Referenced Security BCP in implicit flow attacks where possible

*Minor typo corrections

-02

*Rewrote overview section incorporating feedback from Leo Tohill

*Updated summary recommendation bullet points to split out application and server requirements

*Removed the allowance on hostname-only redirect URI matching, now requiring exact redirect URI matching

*Updated Section 6.2 to drop reference of SPA with a backend component being a public client

*Expanded the architecture section to explicitly mention three architectural patterns available to JS apps

-01

*Incorporated feedback from Torsten Lodderstedt

*Updated abstract

*Clarified the definition of browser-based apps to not exclude applications cached in the browser, e.g. via Service Workers

*Clarified use of the state parameter for CSRF protection

*Added background information about the original reason the implicit flow was created due to lack of CORS support

*Clarified the same-domain use case where the SPA and API share a cookie domain

*Moved historic note about the fragment URL into the Overview

Appendix C. Acknowledgements

The authors would like to acknowledge the work of William Denniss and John Bradley, whose recommendation for native apps 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:

Annabelle Backman, Brian Campbell, Brock Allen, Christian Mainka, Daniel Fett, Eva Sarafianou, George Fletcher, Hannes Tschofenig, Janak Amarasena, John Bradley, Joseph Heenan, Justin Richer, Karl McGuinness, Karsten Meyer zu Selhausen, Leo Tohill, Mike Jones, Philippe De Ryck, Sean Kelleher, Thomas Broyer Tomek Stojecki, Torsten Lodderstedt, Vittorio Bertocci and Yannick Majoros.

Authors' Addresses

Aaron Parecki
Okta

Email: aaron@parecki.com
URI: https://aaronparecki.com

David Waite
Ping Identity

Email: david@alkaline-solutions.com