Open Authentication Protocol
Internet-Draft
Intended status: Best Current Practice
Expires: October 7, 2020

A. Parecki
Okta
D. Waite
Ping Identity

April 05, 2020

OAuth 2.0 for Browser-Based Apps
draft-ietf-oauth-browser-based-apps-06

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.

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 October 7, 2020.

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 running in a browser. This is primarily focused on OAuth, except where OpenID Connect provides additional considerations.

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".

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 9.8 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 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.

Browser-based applications MUST:

- Use the OAuth 2.0 authorization code flow with the PKCE extension
- Protect themselves against CSRF attacks by 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
- Register one or more redirect URIs, and use only exact registered redirect URIs in authorization requests

OAuth 2.0 authorization servers MUST:

- Require exact matching of registered redirect URIs
- Support the PKCE extension
- If issuing refresh tokens to browser-based apps, then:
- Rotate refresh tokens on each use
- Set a maximum lifetime on refresh tokens or expire if they are not used in some amount of 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 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 Grant MUST NOT be used, as described in [oauth-security-topics] section 3.4. 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 Password grant does not provide any built-in mechanism for these, and would instead be extended with custom code.

6. Application Architecture Patterns

There are three primary architectural patterns available when building browser-based applications.

- a JavaScript application that has methods of sharing data with resource servers, such as using common-domain cookies
- a JavaScript application with a backend
- a JavaScript application with no backend, accessing resource servers directly

These three architectures have different use cases and considerations.

6.1. Browser-Based Apps that Can Share Data with the Resource Server

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), OAuth adds additional attack vectors that could be avoided with a different solution.

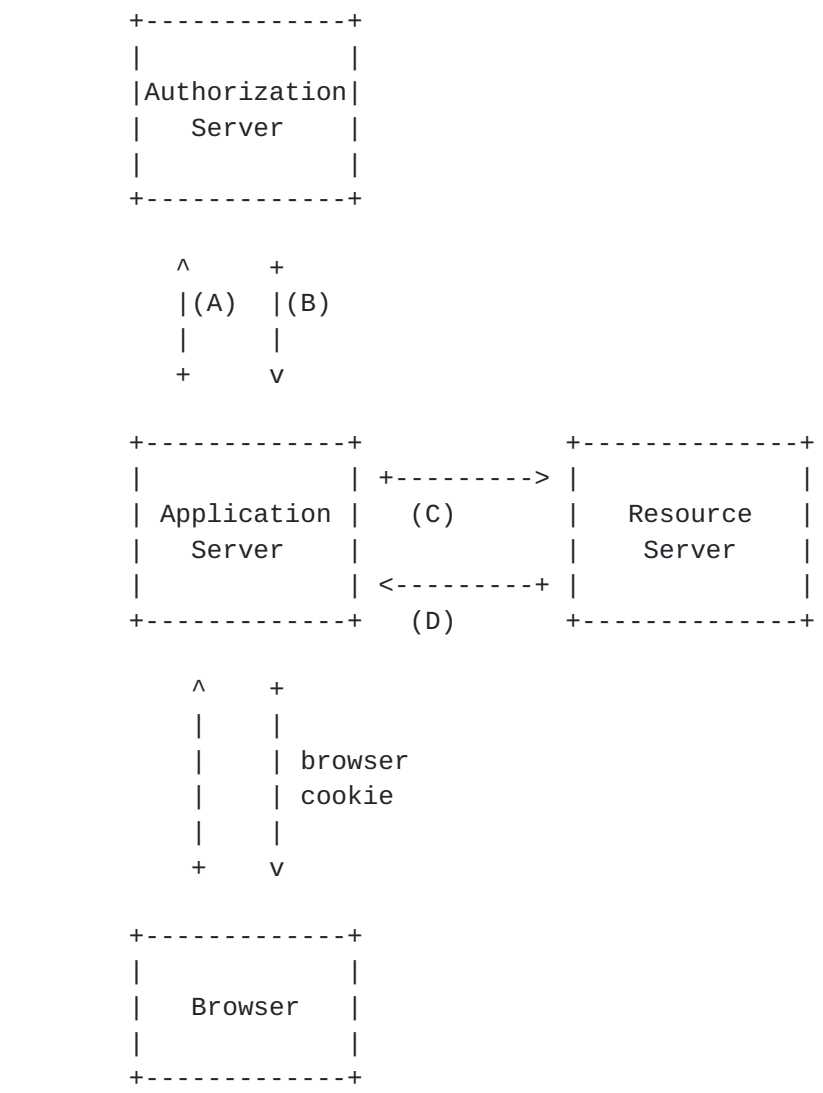
In particular, using any redirect-based mechanism of obtaining an access token enables the redirect-based attacks described in [oauth-security-topics], but if the application, authorization server and resource server share a domain, then it is unnecessary to use a redirect mechanism to communicate between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

An additional concern with handling access tokens in a browser is that as of the date of this publication, there is no secure storage mechanism where JavaScript code can keep the access token to be later used in an API request. Using an OAuth flow results in the JavaScript code getting an access token, needing to store it somewhere, and then retrieve it to make an API request.

Instead, a more secure design is to use an HTTP-only cookie between the JavaScript application and API so that the JavaScript code can't access the cookie value itself. Additionally, the SameSite cookie attribute can be used to prevent CSRF attacks, or alternatively, the application and API could be written to use anti-CSRF tokens.

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, using OAuth even in a common-domain architecture does mean you can more easily rearchitect things later, such as if you were to later add a new domain to the system.

6.2. JavaScript Applications with a Backend



In this architecture, the JavaScript code is loaded from a dynamic Application Server that also has the ability to execute code itself. This enables the ability to keep all of the steps involved in obtaining an access token outside of the JavaScript application.

In this case, the Application Server performs the OAuth flow itself, and keeps the access token and refresh token stored internally, creating a separate session with the browser-based app via a traditional browser cookie.

(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 Application Server SHOULD be considered a confidential client, and issued its own client secret. The Application Server SHOULD use the OAuth 2.0 authorization code grant to initiate a request for an

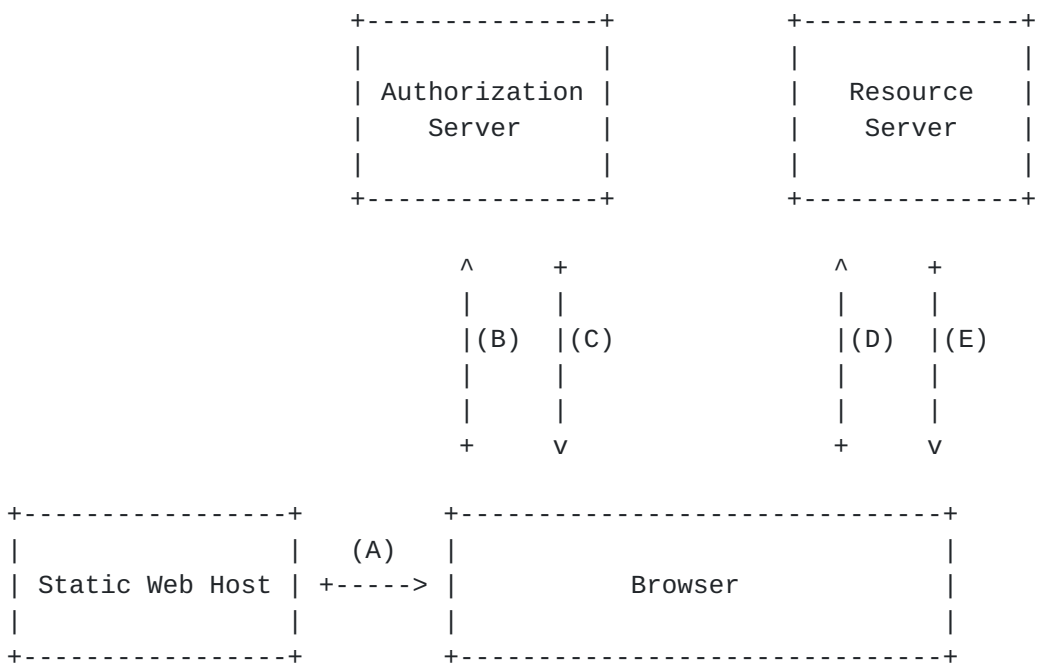
access token. Upon handling the redirect from the Authorization Server, the Application Server will request an access token using the authorization code returned (A), which will be returned to the Application Server (B). The Application Server stores this access token itself and establishes its own cookie-based session with the Browser application. The Application Server can store the access token either server-side, or in the cookie itself.

When the JavaScript application in the browser wants to make a request to the Resource Server, it MUST instead make the request to the Application Server, and the Application Server will make the request with the access token to the Resource Server (C), and forward the response (D) back to the browser.

Security of the connection between code running in the browser and this Application Server 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 Application Server.

In this scenario, the session between the browser and Application Server MAY be either a session cookie provided by the Application Server, OR the access token itself. Note that if the access token is used as the session identifier, this exposes the access token to the end user even if it is not available to the JavaScript application, so some authorization servers may wish to limit the capabilities of these clients to mitigate risk.

6.3. JavaScript Applications without a Backend



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 and there is no other secure client authentication mechanism available in the browser.

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 JavaScript app is then responsible for storing the access token (and optional refresh token) securely using appropriate browser APIs.

When the JavaScript application in the browser wants to make a request to the Resource Server, it can include the access token in the request (D) and make the request directly.

In this scenario, the Authorization Server and Resource Server MUST support the necessary CORS headers to enable the JavaScript code to make this POST request from the domain on which the script is executing. (See Section 9.6 for additional details.)

7. Authorization Code Flow

Public browser-based apps that 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

Public browser-based apps MUST implement the Proof Key for Code Exchange (PKCE [RFC7636]) extension to OAuth, and authorization servers MUST support 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 same client instance that exchanges the authorization code is the same one that initiated the flow.

Browser-based apps 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
- if the application is using OpenID Connect, by using the OpenID Connect "nonce" parameter
- using a unique value for the OAuth 2.0 "state" parameter

Browser-based apps MUST follow the recommendations in [oauth-security-topics] Section 2.1 to protect themselves during redirect flows.

7.2. Handling the Authorization Code Redirect

Authorization servers MUST require an exact match of a registered redirect URI.

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.

Browser-based applications provide an attacker with several opportunities by which a refresh token can be leaked, just as with access tokens. As such, these applications 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 apps.

In particular, authorization servers:

- MUST rotate refresh tokens on each use, in order to be able to detect a stolen refresh token if one is replayed (described in [oauth-security-topics] section 4.12)
- 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.

9. Security Considerations

9.1. Registration of Browser-Based Apps

Browser-based applications 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.

9.2. 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 request 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 9.3 below).

9.3. 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.

9.4. Cross-Site Request Forgery Protections

Clients MUST prevent Cross-Site Request Forgery (CSRF) attacks against their redirect URI. Clients can accomplish this by either ensuring the authorization server supports PKCE and relying on the CSRF protection that PKCE provides, or if the client is also an OpenID Connect client, using the OpenID Connect "nonce" parameter, or by using the "state" parameter to carry one-time-use CSRF tokens as described in Section 7.1.

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

9.5. Authorization Server Mix-Up Mitigation

The security considerations around the authorization server mix-up that are referenced in Section 8.10 of [RFC8252] also apply to browser-based apps.

Clients MUST use a unique redirect URI for each authorization server used by the application. The client MUST store the redirect URI along with the session data (e.g. along with "state") and MUST verify that the URI on which the authorization response was received exactly matches.

9.6. 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.

9.7. 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 ([CSP2]) or similar mechanism. A strong Content Security Policy can limit the potential attack vectors for malicious JavaScript to be executed on the page.

9.8. OAuth Implicit Flow

The OAuth 2.0 Implicit flow (defined in Section 4.2 of OAuth 2.0 [RFC6749]) works by receiving an access token in the HTTP redirect (front-channel) immediately 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.

9.8.1. Attacks on the Implicit Flow

Many attacks on the implicit flow described by [RFC6819] and [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.

9.8.1.1. Threat: Interception 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].

9.8.1.2. 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].

9.8.1.3. 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 man-in-the-middled by, for example, a corporate network. While this type of attack is typically out of scope of basic security recommendations to prevent, in the case of browser-based apps it is much easier to perform this kind of attack, where an injected script can suddenly 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.

9.8.1.4. 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.

9.8.2. Countermeasures

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

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

9.8.3. 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.

9.8.4. Historic Note

Historically, the Implicit flow provided an advantage to single-page 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.

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. 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.

9.9. 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.

10. IANA Considerations

This document does not require any IANA actions.

11. References

11.1. Normative References

[CSP2] West, M., "Content Security Policy", October 2018.

[Fetch] whatwg, "Fetch", 2018.

[oauth-security-topics] Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", July 2019.

[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>.

11.2. Informative References

[HTML] whatwg, "HTML", 2020.

Appendix A. Server Support Checklist

OAuth authorization servers that support browser-based apps MUST:

1. Require "https" scheme redirect URIs.

2. Require exact matching of registered redirect URIs.

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

4. Support cross-domain requests at the token endpoint in order to allow browsers to make the authorization code exchange request. See Section 9.6

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

6. Not support the Resource Owner Password grant for browser-based clients.

7. 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]]

-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, George Fletcher, Hannes Tschofenig, Janak Amarasena, John Bradley, Joseph Heenan, Justin Richer, Karl McGuinness, Leo Tohill, Mike Jones, Tomek Stojecki, Torsten Lodderstedt, and Vittorio Bertocci.

Authors' Addresses

Aaron Parecki
Okta

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

David Waite
Ping Identity

Email: david@alkaline-solutions.com