Open Authentication Protocol
Internet-Draft
Intended status: Best Current Practice
Expires: June 11, 2019

A. Parecki
Okta
D. Waite
Ping Identity

December 08, 2018

OAuth 2.0 for Browser-Based Apps
draft-parecki-oauth-browser-based-apps-02

Abstract

OAuth 2.0 authorization requests from apps running entirely in a browser are unable to use a Client Secret during the process, since they have no way to keep a secret confidential. This specification details the security considerations that must be taken into account when developing browser-based applications, as well as best practices for how they can securely implement 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 June 11, 2019.

Table of Contents

1. Introduction 2
2. Notational Conventions 3
3. Terminology 3
4. Overview 3
5. First-Party Applications 4
6. Architectural Considerations 5
6.1. Apps Served from the Same Domain as the API 5
6.2. Browser-Based App with a Backend Component 5
7. Authorization Code Flow 6
7.1. Initiating the Authorization Request from a Browser-Based Application 6
7.2. Handling the Authorization Code Redirect 6
8. Refresh Tokens 7
9. Security Considerations 7
9.1. Registration of Browser-Based Apps 7
9.2. Client Authentication 7
9.3. Client Impersonation 8
9.4. Cross-Site Request Forgery Protections 8
9.5. Authorization Server Mix-Up Mitigation 8
9.6. Cross-Domain Requests 9
9.7. Content-Security Policy 9
9.8. OAuth Implicit Grant Authorization Flow 9
9.8.1. Threat: Interception of the Redirect URI 10
9.8.2. Threat: Access Token Leak in Browser History 10
9.8.3. Threat: Manipulation of Scripts 10
9.8.4. Threat: Access Token Leak to Third Party Scripts . . 10
9.8.5. Countermeasures 11
9.8.6. Disadvantages of the Implicit Flow 11
9.8.7. Historic Note 12
9.9. Additional Security Considerations 12
10. IANA Considerations 12
11. References 12
11.1. Normative References 12
11.2. Informative References 13
Appendix A. Server Support Checklist 13
Appendix B. Acknowledgements 13
Authors' Addresses 14

1. Introduction

This specification describes the current best practices for implementing OAuth 2.0 authorization flows in applications running entirely 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.

AppAuth steers developers away from performing user authorization via embedding user agents such as browser controls into native apps, instead insisting that an external agent (such as the system browser) be used. The RFC continues on to promote capabilities and supplemental specifications beyond the base OAuth 2.0 and OpenID Connect specifications to improve baseline security, such as [RFC7636], also known as PKCE.

OAuth 2.0 for Browser-Based Apps addresses the similarities between implementing OAuth for native apps as well as 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].

"Browser-based application": An application that runs entirely in a web browser, usually written in JavaScript, where the source code is downloaded from a domain prior to execution. Also sometimes referred to as a "single-page application", or "SPA".

4. Overview

For authorizing users within a browser-based application, the best current practice is to

- Use the OAuth 2.0 authorization code flow with the PKCE extension

- Require the OAuth 2.0 state parameter

- Recommend exact matching of redirect URIs, and require the hostname of the redirect URI match the hostname of the URL the app was served from

- Do not return access tokens in the front channel

Previously it was recommended that browser-based applications use the OAuth 2.0 Implicit flow. That approach has several drawbacks, including the fact that access tokens are returned in the front-channel via the fragment part of the redirect URI, and as such are vulnerable to a variety of attacks where the access token can be intercepted or stolen. See Section 9.8 for a deeper analysis of these attacks and the drawbacks of using the Implicit flow in browsers, many of which are described by [oauth-security-topics].

Instead, browser-based apps can perform the OAuth 2.0 authorization code flow and make a POST request to the token endpoint to exchange an authorization code for an access token, just like other OAuth clients. This ensures that access tokens are not sent via the less secure front-channel, and are only returned over an HTTPS connection initiated from the application. Combined with PKCE, this enables the authorization server to ensure that authorization codes are useless even if intercepted in transport.

5. First-Party Applications

While OAuth and OpenID Connect were initially created to allow third-party applications to access an API on behalf of a user, they have both proven to be useful in a first-party scenario as well. First-party apps are applications created by the same organization that provides the API being accessed by the application.

For example, 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 an OAuth Authorization Code flow as described later in this document or use the OAuth Password grant.

It is strongly RECOMMENDED that applications use the Authorization Code flow over the Password grant for several reasons. By redirecting 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 must be extended with custom code.

6. Architectural Considerations

In some cases, it may make sense to avoid the use of a strictly browser-based OAuth application entirely, instead using an architecture that can provide better security.

6.1. Apps Served from the Same Domain as the API

For simple system architectures, such as when the JavaScript application is served from the same domain as the API (resource server) being accessed, it is likely a better decision to avoid using OAuth entirely, and just use session authentication to communicate with the API.

OAuth and OpenID Connect provide very little benefit in this deployment scenario, so it is recommended to reconsider whether you need OAuth or OpenID Connect at all in this case. Session authentication has the benefit of having fewer moving parts and fewer attack vectors. OAuth and OpenID Connect were created primarily for third-party or federated access to APIs, so may not be the best solution in a same-domain scenario.

6.2. Browser-Based App with a Backend Component

To avoid the risks inherent in handling OAuth access tokens from a purely browser-based application, implementations may wish to move the authorization code exchange and handling of access and refresh tokens into a backend component.

The backend component essentially becomes a new authorization server for the code running in the browser, issuing its own tokens (e.g. a session cookie). Security of the connection between code running in the browser and this backend component is assumed to utilize browser-level protection mechanisms. Details are out of scope of this document, but many recommendations can be found at the OWASP Foundation (https://www.owasp.org/).

In this scenario, the backend component may be a confidential client which is issued its own client secret. Despite this, there are still some ways in which this application is effectively a public client, as the end result is the application's code is still running in the browser and visible to the user. Some authorization servers may have

different policies for public and confidential clients, and this type of hybrid approach does not provide all the assurances of confidential clients that an authorization server is expecting. Authorization servers may wish to treat this type of deployment as a public client.

7. Authorization Code Flow

Public browser-based apps needing user authorization create an authorization request URI with the authorization code grant type per Section 4.1 of OAuth 2.0 [RFC6749], using a redirect URI capable of being received by the app.

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 use the OAuth 2.0 "state" parameter to protect themselves against Cross-Site Request Forgery and authorization code swap attacks and MUST use a unique value for each authorization request, and MUST verify the returned state in the authorization response matches the original state the app created.

7.2. Handling the Authorization Code Redirect

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

If an authorization server wishes to provide some flexibility in redirect URI usage to clients, it MAY require that only the hostname component of the redirect URI match the hostname of the URL the application is served from.

Authorization servers MUST support one of the two redirect URI validation mechanisms as described above.

8. Refresh Tokens

Refresh tokens provide a way for applications to obtain a new access token when the initial access token expires. [oauth-security-topics] describes some additional requirements around refresh tokens on top of the recommendations of [RFC6749].

For public clients, the risk of a leaked refresh token is much greater than leaked access tokens, since an attacker can potentially continue using the stoken refresh token to obtain new access without being detectable by the authorization server. Additionally, browser-based applications provide many attack vectors by which a refresh token can be leaked. As such, these applications are considered a higher risk for handling refresh tokens.

Authorization servers SHOULD NOT issue refresh tokens to browser-based applications.

If an authorization server does choose to issue refresh tokens to browser-based applications, then it MUST issue a new refresh token with every access token refresh response. Doing this mitigates the risk of a leaked refresh token, as a leaked refresh token can be detected if both the attacker and the legitimate client attempt to use the same refresh token. Authorization servers MUST follow the additional refresh token replay mitigation techniques described in [oauth-security-topics].

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. Even when the user has previously approved an authorization request for a given client_id, the request SHOULD be processed as if no previous request had been approved, unless the identity of the client can be proven.

If authorization servers restrict redirect URIs to a fixed set of absolute HTTPS URIs without wildcard domains, paths, or 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

Section 5.3.5 of [RFC6819] recommends using the "state" parameter to link client requests and responses to prevent CSRF (Cross-Site Request Forgery) attacks. To conform to this best practice, use of the "state" parameter is REQUIRED, as described in Section 7.1.

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. If POSTs in particular from unsupported single-page applications are to be rejected as errors per authorization server security policy, such rejection is typically done based on the Origin request header.

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 Grant Authorization Flow

The OAuth 2.0 Implicit grant authorization 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. Several attacks on the implicit flow are described by [RFC6819] and [oauth-security-topics], not all of which have sufficient mitigation strategies.

9.8.1. Threat: Interception of the Redirect URI

If an attacker is able to cause the authorization response to be sent to a URI under his control, he will directly get access to the fragment carrying the access token. A method of performing this attack is described in detail in [oauth-security-topics].

9.8.2. 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.

9.8.3. Threat: Manipulation of Scripts

An attacker could modify the page or inject scripts into the browser via 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 is far greater when the application uses a known standard way of obtaining access tokens, namely that the attacker can always look at the window.location to find an access token. This threat profile is very different compared to 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.

9.8.4. 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.

9.8.5. Countermeasures

In addition to the countermeasures described by [RFC6819] and [oauth-security-topics], using the authorization code with PKCE avoids these attacks.

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

9.8.6. 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 an access token was issued to it, 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 little 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. Performing OpenID Connect using the authorization code flow also provides the additional benefit of the client not needing to verify the JWT signature, as the token will have been fetched over an HTTPS connection directly from the authorization server. However, returning an id_token using the Implicit flow requires the client validate the JWT signature, as malicious parties could otherwise craft and supply fraudulent id_tokens.

9.8.7. 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. Now with the Session History API (described in "Session history and navigation" of [HTML]), browsers have a mechanism to modify the path component of the URL without triggering a page reload, so this overloaded use of the fragment portion is no longer needed.

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., Barth, A., and D. Veditz, "Content Security Policy", December 2016.

[Fetch] whatwg, "Fetch", 2018.

[oauth-security-topics] Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", November 2018.

[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>.

[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", 2018.

Appendix A. Server Support Checklist

OAuth servers that support browser-based apps MUST:

1. Require "https" scheme redirect URIs.

2. Require exact matching on redirect URIs or matching the hostname the application is served from.

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.

Appendix B. 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, as well as all the attendees of the Internet Identity Workshop 27 session at which this BCP was originally proposed.

The following individuals 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, John Bradley, Joseph Heenan, Justin Richer, Karl McGuinness, 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