Network Working Group
Internet-Draft
Intended status: Experimental
Expires: May 6, 2019

A. Parecki
Okta
D. Waite
Ping Identity

November 02, 2018

OAuth 2.0 for Browser-Based Apps
draft-parecki-oauth-browser-based-apps-00

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 May 6, 2019.

Copyright Notice

Table of Contents

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.

This specification 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, 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 match the hostname of the URL the app was served from
- Do not return access tokens in the front channel

Each of these is described in more detail in the sections below.

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 applicaiton.

For example, an 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.

5.1. 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. 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.

6.1. 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.

6.2. 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.

7. Security Considerations

7.1. 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.

7.2. Client Authentication

Similar to native apps, a browser-based with native OAuth support is a public client. Since the application source code is delivered to the end-user's browser, it cannot contain provisioned secrets.

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 (as defined by Section 2.1 of OAuth 2.0 [RFC6749]), and not accept the secret as proof of the client's identity. Without additional measures, such clients are subject to client impersonation (see Section 7.3 below).

7.3. 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 or paths, 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.

7.4. 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 6.1.

7.5. 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.

7.6. 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 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.

7.7. 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.

7.8. 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. The Implicit Flow cannot be protected by PKCE [RFC7636] (which is required according to Section 6), so clients and authorization servers MUST NOT use the Implicit Flow for browser-based apps.

There are several 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.

- 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 simplifies the implementation.

- If the JavaScript application gets wrapped into a native app, then [RFC8252] also requires the use of the authorization code flow.

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.

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.

7.9. 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.

8. IANA Considerations

This document does not require any IANA actions.

9. References

9.1. Normative References

[CSP2] West, M., Barth, A., and D. Veditz, "Content Security Policy", December 2016.

[Fetch] whatwg, ., "Fetch", 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>.

9.2. Informative References

[HTML] whatwg, ., "HTML", 2018.

Appendix A. 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 6.1

4. Support cross-domain requests at the token endpoint in order to allow browsers to make the authorization code exchange request. See Section 7.6

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 acknowledge Hannes Tschofenig as well as all the attendees of the Internet Identity Workshop 27 session at which this BCP was originally proposed.

Authors' Addresses

Aaron Parecki
Okta

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

David Waite
Ping Identity

Email: david@alkaline-solutions.com