OAuth 2.0 for Browser-Based Apps
draft-ietf-oauth-browser-based-apps-17

Abstract

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

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 31 August 2024.

Copyright Notice

Table of Contents

1. Introduction

   This specification describes different architectural patterns for
   implementing OAuth 2.0 in applications executing in a browser. The
   specification outlines the security challenges for browser-based
   applications and analyzes how different patterns address these
   challenges.

   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, but also
highlights how the security properties of browser-based applications
are vastly different than those of native applications. This
document 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 document
expands on and further restricts various recommendations given 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 applications", this
is not intended to be exclusive to the JavaScript language. The
recommendations and considerations herein also apply to other
languages that execute code in the browser, such as Web Assembly
(https://webassembly.org/).

4. History of OAuth 2.0 in Browser-Based Applications

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 7.2 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, tokens are no longer exposed in the less-secure front channel, which makes the use of refresh tokens possible for browser-based applications. 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.

5. The Threat of Malicious JavaScript

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

When analyzing the security of browser-based applications in light of the presence of malicious JS, it is crucial to realize that the *malicious JavaScript code has the same privileges as the legitimate

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

This section explores the threats malicious JS code poses to browser-
based applications that assume the role of an OAuth client. The
first part discusses a few scenarios that attackers can use once they
found a way to run malicious JavaScript code. These scenarios paint
a clear picture of the true power of the attacker, which goes way
beyond simple token exfiltration. The second part of this section
analyzes the impact of these attack scenarios on the OAuth client.

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

5.1. Malicious JavaScript Payloads

This section presents several malicious scenarios that an attacker
can execute once they have found a vulnerability that allows the
execution of malicious JavaScript code. The attack scenarios range
from extremely trivial (Section 5.1.1) to highly sophisticated
(Section 5.1.3). Note that this enumeration is non-exhaustive and
presented in no particular order.

5.1.1. Single-Execution Token Theft

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

* Execute malicious JS code

* Obtain tokens from the application's preferred storage mechanism
   (See Section 8)

* Send the tokens to a server controlled by the attacker

* Store/abuse the stolen tokens

The recommended defensive strategy to protect access tokens is to reduce the scope and lifetime of the token. For refresh tokens, the use of refresh token rotation offers a detection and correction mechanism. Sender-constrained tokens (Section 9.2) offer an additional layer of protection against stolen access tokens.

Note that this attack scenario is trivial and often used to illustrate the dangers of malicious JavaScript. Unfortunately, it significantly underestimates the capabilities of a sophisticated and motivated attacker.

5.1.2. Persistent Token Theft

This attack scenario is a more advanced variation on the Single-Execution Token Theft scenario (Section 5.1.1). Instead of immediately stealing tokens upon the execution of the payload, the attacker sets up the necessary handlers to steal the application's tokens on a continuous basis. This scenario consists of the following steps:

* Execute malicious JS code

* Setup a continuous token theft mechanism (e.g., on a 10-second time interval) - Obtain tokens from the application's preferred storage mechanism (See Section 8) - Send the tokens to a server controlled by the attacker - Store the tokens

* Wait until the opportune moment to abuse the latest version of the stolen tokens

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

For access tokens, the attacker now obtains the latest access token for as long as the user's browser is online. Refresh token rotation is not sufficient to prevent abuse of a refresh token. An attacker can easily wait until the user closes the application or their browser goes offline before using the latest refresh token, thereby ensuring that the latest refresh token is not reused.

5.1.3. Acquisition and Extraction of New Tokens

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

   * Execute malicious JS code

   * Setup a handler to obtain the authorization code from the iframe
      (e.g., by monitoring the frame's URL or via Web Messaging)

   * Insert a hidden iframe into the page and initialize it with an
      authorization request. The authorization request in the iframe
      will occur within the user's session and, if the session is still
      active, result in the issuing of an authorization code.

   * Extract the authorization code from the iframe using the
      previously installed handler

   * Send the authorization code to a server controlled by the attacker

   * Exchange the authorization code for a new set of tokens

   * Abuse the stolen tokens

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

   This attack scenario is possible because the security of public
   browser-based OAuth 2.0 clients relies entirely on the redirect URI
   and application's origin. When the attacker executes malicious
   JavaScript code in the application's origin, they gain the capability
   to inspect same-origin frames. As a result, the attacker's code
   running in the main execution context can inspect the redirect URI
   loaded in the same-origin frame to extract the authorization code.

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

5.1.4. Proxying Requests via the User's Browser

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

* Execute malicious JS code

* Send a request to a resource server and process the response

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

This attack pattern is well-known and also occurs with traditional
applications using HttpOnly session cookies. It is commonly accepted
that this scenario cannot be stopped or prevented by application-
level security measures. For example, the DPoP specification
([DPoP]) explicitly considers this attack scenario to be out of
scope.

5.2. Attack Consequences

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

5.2.1. Exploiting Stolen Refresh Tokens

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

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

5.2.2. Exploiting Stolen Access Tokens

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

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

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

   The application can use DPoP to ensure its access tokens are bound to
   non-exportable keys held by the browser. In that case, it becomes
   significantly harder for the attacker to abuse stolen access tokens.
   More specifically, with DPoP, the attacker can only abuse stolen
   application tokens by carrying out an online attack, where the proofs

are calculated in the user's browser. This attack is described in
detail in section 11.4 of the [DPoP] specification. Additionally,
when the attacker obtains a fresh set of tokens, as described in
Section 5.1.3, they can set up DPoP for these tokens using an
attacker-controlled key pair. In that case, the attacker is again
free to abuse this newly obtained access token without restrictions.

5.2.3. Client Hijacking

When stealing tokens is not possible or desirable, the attacker can
also choose to hijack the OAuth client application running in the
user's browser. This effectively allows the attacker to perform any
operations that the legitimate client application can perform.
Examples include inspecting data on the page, modifying the page, and
sending requests to backend systems.

Note that client hijacking is less powerful than directly abusing
stolen tokens. In a client hijacking scenario, the attacker cannot
directly control the tokens and is restricted by the security
policies enforced on the client application. For example, a resource
server running on admin.example.org can be configured with a Cross-
Origin Resource Sharing (CORS) policy that rejects requests coming
from a client running on web.example.org. Even if the access token
used by the client would be accepted by the resource server, the CORS
configuration does not allow such a request.

6. Application Architecture Patterns

There are three main architectural patterns available when building
browser-based JavaScript applications that rely on OAuth 2.0 for
accessing protected resources.

   * A JavaScript application that relies on a backend component for
      handling OAuth responsibilities and proxies all requests through
      the backend component (Backend-For-Frontend or BFF)

   * A JavaScript application that relies on a backend component for
      handling OAuth responsibilities, but calls resource servers
      directly using the access token (Token-Mediating Backend)

   * A JavaScript application acting as the client, handling all OAuth
      responsibilities in the browser (Browser-based OAuth 2.0 Client)

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

6.1. Backend For Frontend (BFF)

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

   1. The BFF interacts with the authorization server as a confidential
       OAuth client

   2. The BFF manages OAuth access and refresh tokens, making them
       inaccessible by the JavaScript application

   3. The BFF proxies all requests to a resource server, augmenting
       them with the correct access token before forwarding them to the
       resource server

   If an attacker is able to execute malicious code within the
   JavaScript application, the application architecture is able to
   withstand most of the payload scenarios discussed before. Since
   tokens are only available to the BFF, there are no tokens available
   to extract from JavaScript (Payload Section 5.1.1 and Section 5.1.2).
   The BFF is a confidential client, which prevents the attacker from
   running a new flow within the browser (Payload Section 5.1.3). Since
   the malicious JavaScript code still runs within the application's
   origin, the attacker is able to send requests to the BFF from within
   the user's browser (Payload Section 5.1.4).

6.1.1. Application Architecture
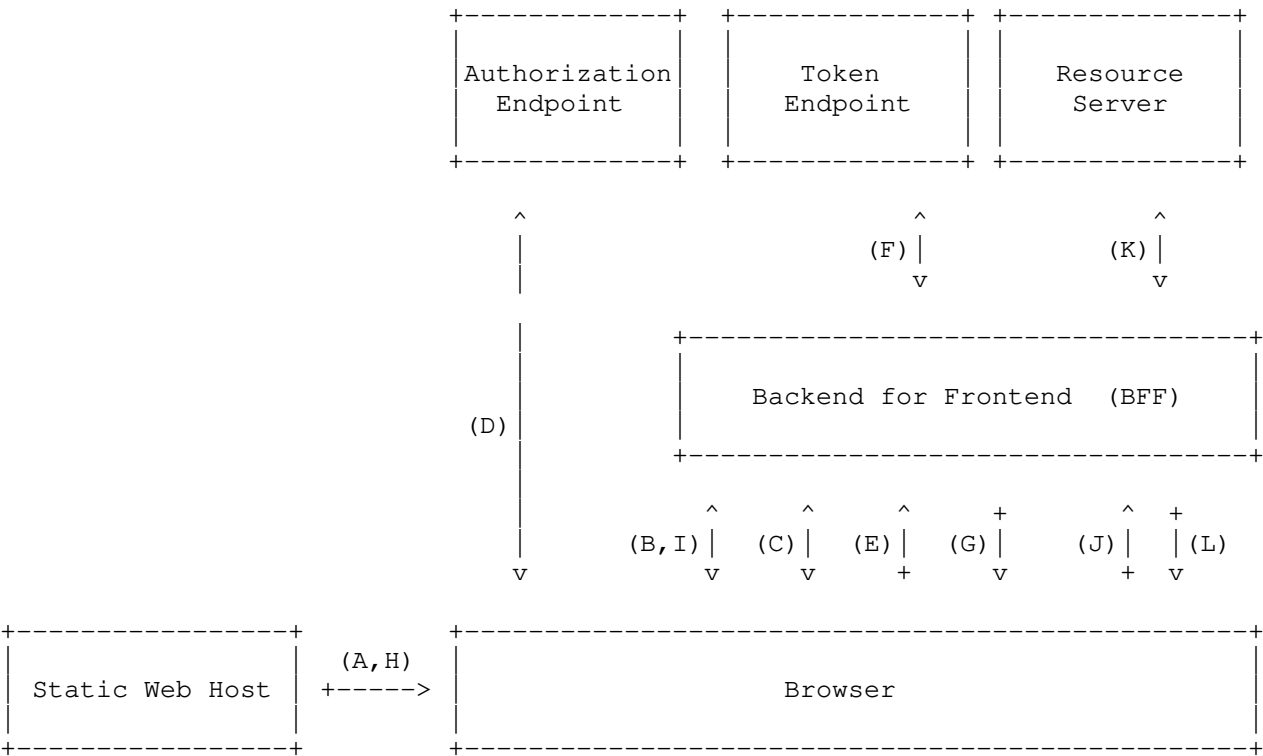
```
         +-------------+  +-------------+ +-------------+
         |             |  |             | |             |
         |Authorization|  |   Token     | |  Resource   |
         |  Endpoint   |  |  Endpoint   | |   Server    |
         |             |  |             | |             |
         +-------------+  +-------------+ +-------------+
               ^                 ^               ^
               |             (F) |           (K) |
               |                 v               v
               |       +----------------------------------+
               |       |                                  |
               |       |    Backend for Frontend  (BFF)   |
           (D) |       |                                  |
               |       +----------------------------------+
               |
               |        ^      ^      ^      +      ^      +
               |  (B,I) |  (C) |  (E) |  (G) |  (J) |   | (L)
               v        v      v      +      v      +   v

+----------------+     +-------------------------------------------+
|                |(A,H)|                                           |
| Static Web Host|+--->|                Browser                    |
|                |     |                                           |
+----------------+     +-------------------------------------------+
```

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. The application checks with the BFF if there is an
active session (B). If an active session is found, the application
resumes its authenticated state and skips forward to step J.

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

The BFF associates the obtained tokens with the user's session (See Section 6.1.2.2) and includes the relevant information in a cookie that is included in the response to the browser (G). This response to the browser will also trigger the reloading of the JavaScript application (H). When this application reloads, it will check with the BFF for an existing session (I), allowing the JavaScript application to resume its authenticated state.

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

6.1.2. Implementation Details

6.1.2.1. Refresh Tokens

It is recommended to use both access tokens and refresh tokens, as it enables access tokens to be short-lived and minimally scoped (e.g., using [RFC8707]). When using refresh tokens, the BFF obtains the refresh token in step F and associates it with the user's session.

If the BFF notices that the user's access token has expired and the BFF has a refresh token, it can run a Refresh Token flow to obtain a fresh access token. These steps are not shown in the diagram, but would occur between step J and K. Note that this BFF client is a confidential client, so it will use its client authentication in the Refresh Token request.

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

6.1.2.2. Cookie-based Session Management

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

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

   Client-side sessions push all data to the browser in a signed, and
   optionally encrypted, object. This pattern absolves the server of
   keeping track of any session data, but severely limits control over
   active sessions and makes it difficult to handle session revocation.
   However, when client-side sessions are used in the context of a BFF,
   these properties change significantly. Since the cookie-based
   session is only used to obtain a user's tokens, all control and
   revocation properties follow from the use of access tokens and
   refresh tokens. It suffices to revoke the user's access token and/or
   refresh token to prevent ongoing access to protected resources,
   without the need to explicitly invalidate the cookie-based session.

   Best practices to secure the session cookie are discussed in
   Section 6.1.3.2.

6.1.2.3. Combining OAuth and OpenID Connect

   The OAuth flow used by this application architecture can be combined
   with OpenID Connect by including the necessary OpenID Connect scopes
   in the authorization request (C). In that case, the BFF will receive
   an ID Token in step F. The BFF can associate the information from
   the ID Token with the user's session and provide it to the JavaScript
   application in step B or I.

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

6.1.2.4. Practical Deployment Scenarios

   Serving the static JavaScript code is a separate responsibility from
   handling OAuth tokens and proxying requests. In the diagram
   presented above, the BFF and static web host are shown as two
   separate entities. In real-world deployment scenarios, these
   components can be deployed as a single service (i.e., the BFF serving
   the static JS code), as two separate services (i.e., a CDN and a
   BFF), or as two components in a single service (i.e., static hosting
   and serverless functions on a cloud platform).

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

6.1.3. Security Considerations

6.1.3.1. The Authorization Code Flow

The main benefit of using a BFF is the BFF's ability to act as a
confidential client. Therefore, the BFF MUST act as a confidential
client. Furthermore, the BFF 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.

6.1.3.2. Cookie Security

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

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

* The BFF MUST enable the _Secure_ flag for its cookies

* The BFF MUST enable the _HttpOnly_ flag for its cookies

* The BFF SHOULD enable the _SameSite=Strict_ flag for its cookies

* The BFF SHOULD set its cookie path to _/_

* The BFF SHOULD NOT set the _Domain_ attribute for cookies

* The BFF SHOULD start the name of its cookies with the ___Host-_
   prefix ([CookiePrefixes])

Additionally, when using client-side sessions that contain access
tokens, (as opposed to server-side sessions where the tokens only
live on the server), the BFF SHOULD encrypt its cookie contents using
an Authenticated Encryption with Authenticated Data ([RFC5116]).

This ensures that tokens stored in cookies are never written to the user's hard drive in plaintext format. This security measure helps to ensure protection of the access token against malware that actively scans the user's hard drive to extract sensitive browser artifacts, such as cookies and locally stored data (see Section 8).

For further guidance on cookie security best practices, we refer to the OWASP Cheat Sheet series (https://cheatsheetseries.owasp.org (https://cheatsheetseries.owasp.org)).

6.1.3.3. Cross-Site Request Forgery Protections

The interactions between the JavaScript application and the BFF rely on cookies for authentication and authorization. Similar to other cookie-based interactions, the BFF is required to account for Cross-Site Request Forgery (CSRF) attacks.

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

6.1.3.3.1. SameSite Cookie Attribute

Configuring the cookies with the _SameSite=Strict_ attribute (See Section 6.1.3.2) ensures that the BFF's cookies are only included on same-site requests, and not on potentially malicious cross-site requests.

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

For example, subdomains, such as https://a.example.com and https://b.example.com, are considered same-site, since they share the same site example.com. They are considered cross-origin, since origins consist of the tuple _<scheme, hostname, port>_. As a result, a subdomain takeover attack against b.example.com can enable CSRF attacks against the BFF of a.example.com. Technically, this attack should be identified as a "Same-Site But Cross-Origin Request Forgery" attack.

6.1.3.3.2. Cross-Origin Resource Sharing (CORS)

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

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

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

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

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

   To avoid such bypasses against the CORS policy, the BFF SHOULD
   require that every request includes a custom request header. Cross-
   origin requests with a custom request header always require a
   preflight, which makes CORS an effective CSRF defense. Implementing
   this mechanism is as simple as requiring every request to have a
   static request header, such as X-CORS-Security: 1.

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

6.1.3.3.3. Use anti-forgery/double submit cookies

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

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

6.1.3.4. Advanced Security

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

6.1.4. Threat Analysis

   This section revisits the payloads and consequences from Section 5,
   and discusses potential additional defenses.

6.1.4.1. Attack Payloads and Consequences

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

   * Proxying Requests via the User's Browser (See Section 5.1.4)

   Note that this attack scenario results in the following consequences:

   * Client Hijacking (See Section 5.2.3)

   Unfortunately, client hijacking is an attack scenario that is
   inherent to the nature of browser-based applications. As a result,
   nothing will be able to prevent such attacks apart from stopping the
   execution of malicious JavaScript code in the first place.

Techniques that can help to achieve this are following secure coding guidelines, code analysis, and deploying defense-in-depth mechanisms such as Content Security Policy ([CSP3]).

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

6.1.4.2. Mitigated Attack Scenarios

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

* Single-Execution Token Theft (See Section 5.1.1)

* Persistent Token Theft (See Section 5.1.2)

* Acquisition and Extraction of New Tokens (See Section 5.1.3)

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

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

Because of the nature of the BFF, the following two consequences of potential attacks become irrelevant:

* Exploiting Stolen Refresh Tokens (See Section 5.2.1)

* Exploiting Stolen Access Tokens (See Section 5.2.2)

6.1.4.3. Summary

   To summarize, the architecture of a BFF is significantly more
   complicated than a browser-only application. It requires deploying
   and operating a server-side BFF component. Additionally, this
   pattern requires all interactions between the JavaScript application
   and the resource servers to be proxied by the BFF. Depending on the
   deployment pattern, this proxy behavior can add a significant burden
   on the server-side components. See Section 6.1.2.4 for additional
   notes if the BFF is acting as the resource server.

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

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

6.2. Token-Mediating Backend

   This section describes the architecture of a JavaScript application
   that relies on a backend component to handle OAuth responsibilities
   for obtaining tokens, after which the JavaScript application receives
   the access token to directly interact with resource servers.

   The token-mediating backend pattern is more lightweight than the BFF
   pattern (See Section 6.1), since it does not require the proxying of
   all requests to a resource server, which improves latency and
   significantly simplifies deployment. From a security perspective,
   the token-mediating backend is less secure than a BFF, but still
   offers significant advantages over an OAuth client application
   running directly in the browser.

   If an attacker is able to execute malicious code within the
   JavaScript application, the application architecture is able to
   prevent the attacker from abusing refresh tokens or obtaining a fresh
   set of tokens. However, since the access token is directly exposed
   to the JavaScript application, token theft scenarios fall within the
   capabilities of the attacker.

6.2.1. Application Architecture
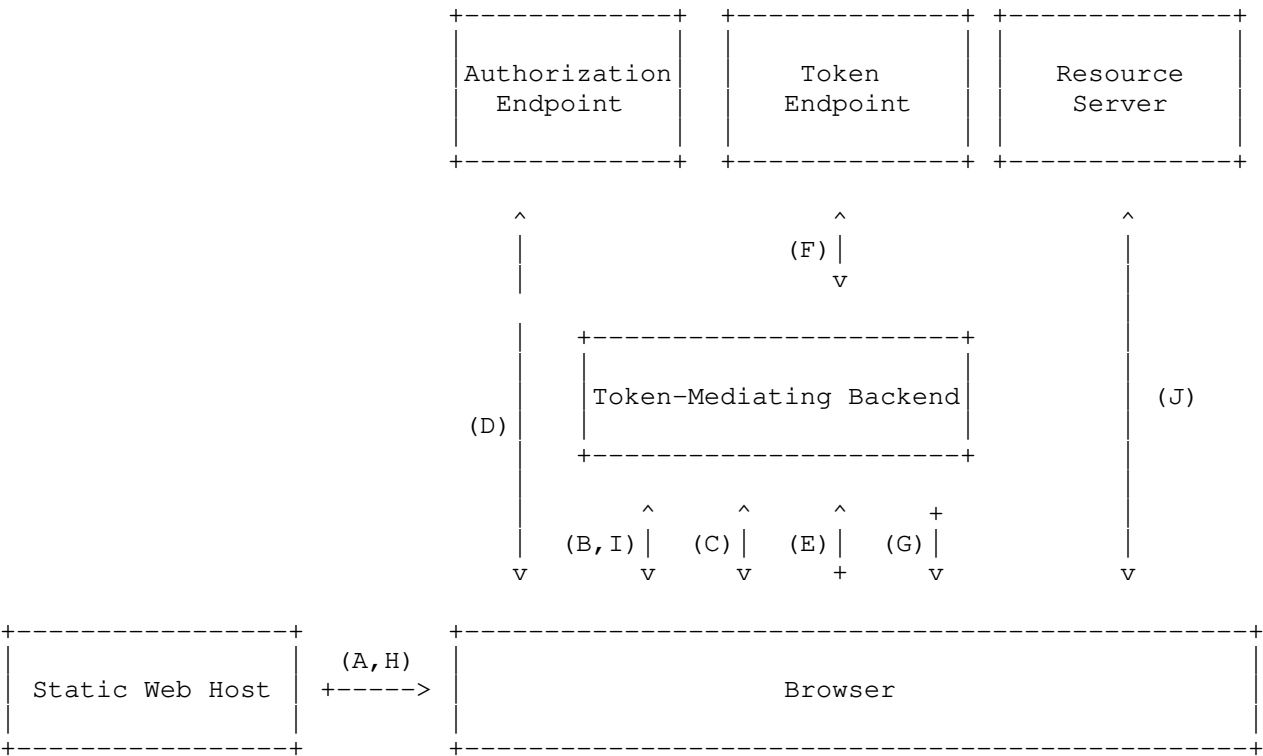
```
                  +-------------+  +-------------+  +-------------+
                  |             |  |             |  |             |
                  |Authorization|  |    Token    |  |  Resource   |
                  |  Endpoint   |  |  Endpoint   |  |   Server    |
                  |             |  |             |  |             |
                  +-------------+  +-------------+  +-------------+

                        ^                ^                ^
                        |            (F) |                |
                        |                v                |
                        |   +---------------------+       |
                        |   |                     |       |
                        |   |Token-Mediating Backend      |   (J)
                   (D)  |   |                     |       |
                        |   +---------------------+       |
                        |                                 |
                        |    ^      ^      ^      +       |
                        | (B,I)|  (C)|  (E)|  (G)|        |
                        v     v      v      v    +    v   v

 +----------------+          +------------------------------------------------+
 |                |  (A,H)    |                                                |
 | Static Web Host|  +----->  |                     Browser                   |
 |                |          |                                                |
 +----------------+          +------------------------------------------------+
```

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. The application checks with the token-mediating backend if there is an active session (B). If an active session is found, the application receives the corresponding access token, resumes its authenticated state, and skips forward to step J.

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

The token-mediating backend associates the obtained tokens with the user's session (See Section 6.2.2.3) and includes the relevant information in a cookie that is included in the response to the browser (G). This response to the browser will also trigger the

reloading of the JavaScript application (H). When this application
reloads, it will check with the token-mediating backend for an
existing session (I), allowing the JavaScript application to resume
its authenticated state and obtain the access token from the token-
mediating backend.

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

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.

6.2.2. Implementation Details

6.2.2.1. Refresh Tokens

It is recommended to use both access tokens and refresh tokens, as it
enables access tokens to be short-lived and minimally scoped (e.g.,
using [RFC8707]). When using refresh tokens, the token-mediating
backend obtains the refresh token in step F and associates it with
the user's session.

If the resource server rejects the access token, the JavaScript
application can contact the token-mediating backend to request a
fresh access token. The token-mediating backend relies on the
cookies associated with this request to use the user's refresh token
to run a Refresh Token flow. These steps are not shown in the
diagram. Note that this Refresh Token flow involves a confidential
client, thus requires client authentication.

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

6.2.2.2. Access Token Scopes

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

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

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

The token-mediating backend can use a similar mechanism to downscoping when relying on [RFC8707] to obtain access token for a specific resource server.

6.2.2.3. Cookie-based Session Management

Similar to the BFF, the token-mediating backend relies on traditional browser cookies to keep track of the user's session. The same implementation guidelines and security considerations as for a BFF apply, as discussed in Section 6.1.2.2.

6.2.2.4. Combining OAuth and OpenID Connect

Similar to a BFF, the token-mediating backend can choose to combine OAuth and OpenID Connect in a single flow. See Section 6.1.2.3 for more details.

6.2.2.5. Practical Deployment Scenarios

Serving the static JavaScript code is a separate responsibility from handling interactions with the authorization server. In the diagram presented above, the token-mediating backend and static web host are shown as two separate entities. In real-world deployment scenarios, these components can be deployed as a single service (i.e., the token-mediating backend serving the static JS code), as two separate services (i.e., a CDN and a token-mediating backend), or as two components in a single service (i.e., static hosting and serverless functions on a cloud platform). These deployment differences do not affect the relationships described in this pattern.

6.2.3. Security Considerations

6.2.3.1. The Authorization Code Grant

   The main benefit of using a token-mediating backend is the backend's
   ability to act as a confidential client. Therefore, the token-
   mediating backend MUST act as a confidential client. Furthermore,
   the token-mediating backend 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.

6.2.3.2. Cookie Security

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

6.2.3.3. Cross-Site Request Forgery Protections

   The interactions between the JavaScript application and the token-
   mediating backend rely on cookies for authentication and
   authorization. Just like a BFF, the token-mediating backend is
   required to account for Cross-Site Request Forgery (CSRF) attacks.

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

6.2.3.4. Advanced OAuth Security

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

6.2.4. Threat Analysis

   This section revisits the payloads and consequences from Section 5,
   and discusses potential additional defenses.

6.2.4.1. Attack Payloads and Consequences

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

* Single-Execution Token Theft (See Section 5.1.1) for access tokens

* Persistent Token Theft (See Section 5.1.2) for access tokens

* Proxying Requests via the User's Browser (See Section 5.1.4)

Note that this attack scenario results in the following consequences:

* Exploiting Stolen Access Tokens (See Section 5.2.2)

* Client Hijacking (See Section 5.2.3)

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

6.2.4.2. Mitigated Attack Scenarios

The other payloads, listed below, are effectively mitigated by the token-mediating backend:

* Single-Execution Token Theft (See Section 5.1.1) for refresh tokens

* Persistent Token Theft (See Section 5.1.2) for refresh tokens

* Acquisition and Extraction of New Tokens (See Section 5.1.3)

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

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

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

* Exploiting Stolen Refresh Tokens (See Section 5.2.1)

6.2.4.3. Additional Defenses

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

6.2.4.3.1. Secure Token Storage

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

   Note that even when the access token is stored out of reach of
   malicious JavaScript code, the attacker still has the ability to
   request the access token from the token-mediating backend.

6.2.4.3.2. Using Sender-Constrained Tokens

   Using sender-constrained access tokens is not trivial in this
   architecture. The token-mediating backend is responsible for
   exchanging an authorization code or refresh token for an access
   token, but the JavaScript application will use the access token.
   Using a mechanism such as [DPoP] would require proof generation for a
   request to the authorization server in the JavaScript application,
   but use of that proof by the token-mediating backend.

6.2.4.4. Summary

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

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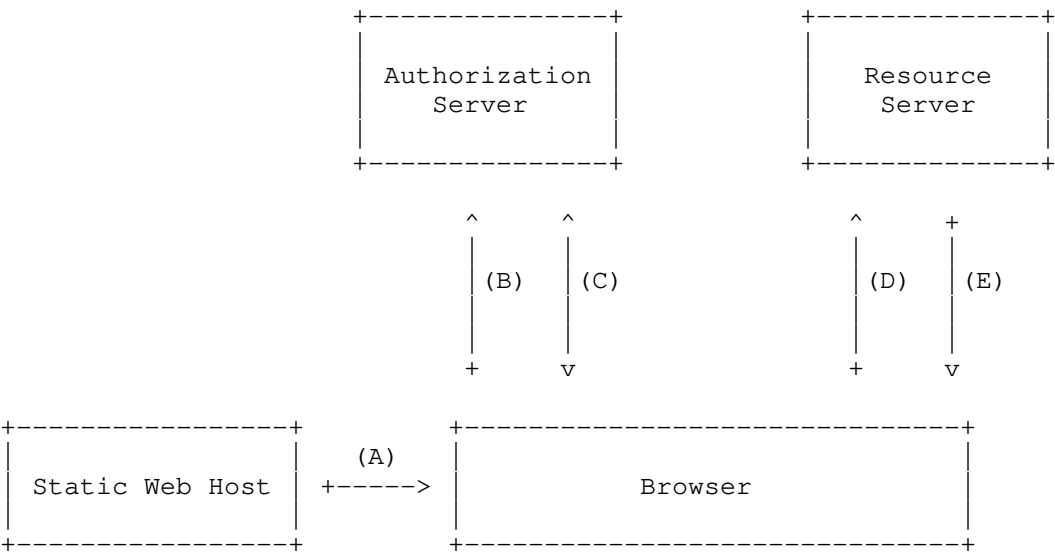   to gain direct access to resource servers.

   When considering a token-mediating backend architecture, it is
   strongly recommended to go the extra mile and adopt a full BFF as
   discussed in Section 6.1. Only when the use cases or system
   requirements would prevent the use of a proxying BFF should the
   token-mediating backend be considered as viable alternative.

6.3. Browser-based OAuth 2.0 client

   This section describes the architecture of a JavaScript application
   that acts as the OAuth 2.0 client, handling all OAuth
   responsibilities in the browser. As a result, the browser-based
   application obtains tokens from the authorization server, without the
   involvement of a backend component.

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

6.3.1. Application Architecture

```
                      +--------------+              +--------------+
                      |              |              |              |
                      | Authorization|              |   Resource   |
                      |    Server    |              |    Server    |
                      |              |              |              |
                      +--------------+              +--------------+

                         ^      ^                      ^       +
                         |      |                      |       |
                         | (B)  | (C)                  | (D)   | (E)
                         |      |                      |       |
                         +      v                      +       v

       +----------------+        +------------------------------+
       |                |  (A)   |                              |
       | Static Web Host| +----> |            Browser           |
       |                |        |                              |
       +----------------+        +------------------------------+
```

   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 provision it with client credentials in this
   model.

   The application obtains an authorization code (B) by initiating the
   Authorization Code flow with the PKCE extension (described in
   Section 6.3.2.1). The application exchanges the authorization code
   for tokens via a JavaScript-based POST request to the token endpoint
   (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 8.

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

6.3.2. Security Considerations

6.3.2.1. The Authorization Code Grant

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.

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

*   MUST use PKCE ([RFC7636]) when obtaining an access token (Section 6.3.2.1.1)

*   MUST Protect themselves against CSRF attacks (Section 6.3.2.6) 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 6.3.2.4.1)

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 6.3.2.4.1)

*   MUST support the PKCE extension (Section 6.3.2.1.1)

*   MUST NOT issue access tokens in the authorization response (Section 7.2)

   * If issuing refresh tokens to browser-based applications
      (Section 6.3.2.7), 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

6.3.2.1.1. 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.

6.3.2.2. Registration of Browser-Based Apps

   Browser-only OAuth clients 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 (See Section 6.3.2.4.1).

   Note that both the BFF and token-mediating backend are confidential
   clients.

6.3.2.3. 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 no value beyond client identification which is already provided by the client_id parameter.

Authorization servers that still require a statically included shared secret for SPA clients MUST treat the client as a public client, and not accept the secret as proof of the client's identity. Without additional measures, such clients are subject to client impersonation (see Section 6.3.2.4 below).

6.3.2.4. 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.

6.3.2.4.1. Authorization Code Redirect

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

Authorization servers MUST require an exact match of a registered redirect URI as described in [oauth-security-topics] Section 4.1.1. This helps to prevent attacks targeting the authorization code.

6.3.2.5. Security of In-Browser Communication Flows

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

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

Section 4.18. of [oauth-security-topics] provides additional details about the security of in-browser communication flows and the countermeasures that browser-based apps and authorization servers MUST apply to defend against these attacks.

6.3.2.6. 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 and verifying 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.

6.3.2.7. Refresh Tokens

Refresh tokens provide a way for applications to obtain a new access token when the initial access token expires. For browser-based clients, the refresh token is typically a bearer token, unless the application explicitly uses [DPoP]. As a result, 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.

Authorization servers may choose whether or not to issue refresh tokens to browser-based applications. However, in light of the impact of third-party cookie blocking mechanisms, the use of refresh tokens has become significantly more attractive. The [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 4.14.2

* 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 10 minutes, and a refresh token that lasts 8 hours

* After 10 minutes, the initial access token expires, so the
  application uses the refresh token to get a new access token

* The authorization server returns a new access token that lasts 10
  minutes, and a new refresh token that lasts 7 hours and 50 minutes

* This continues until 8 hours pass from the initial authorization

* At this point, when the application attempts to use the refresh
  token after 8 hours, the request will fail and the application
  will have to re-initialize an Authorization Code flow that relies
  on the user's authentication or previously established session

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

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

6.3.2.8. Cross-Origin Requests

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

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

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

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

6.3.3. Threat Analysis

   This section revisits the payloads and consequences from Section 5,
   and discusses potential additional defenses.

6.3.3.1. Attack Payloads and Consequences

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

   * Single-Execution Token Theft (See Section 5.1.1)

   * Persistent Token Theft (See Section 5.1.2)

   * Acquisition and Extraction of New Tokens (See Section 5.1.3)

   * Proxying Requests via the User's Browser (See Section 5.1.4)

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

   Note that these attack scenarios result in the following
   consequences:

   * Exploiting Stolen Refresh Tokens (See Section 5.2.1)

   * Exploiting Stolen Access Tokens (See Section 5.2.2)

   * Client Hijacking (See Section 5.2.3)

6.3.3.2. Additional Defenses

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

6.3.3.2.1. Secure Token Storage

   When handling tokens directly, the application can choose different
   storage mechanisms to handle access tokens and refresh tokens.
   Universally accessible storage areas, such as _Local Storage_, are
   easier to access from malicious JavaScript than highly isolated
   storage areas, such as a _Web Worker_. Section 8 discusses different
   storage mechanisms with their trade-off in more detail.

   A practical implementation pattern can use a Web Worker to isolate
   the refresh token, and provide the application with the access token
   making requests to resource servers.

   Note that even a perfect token storage mechanism does not prevent the
   attacker from running a new flow to obtain a fresh set of tokens (See
   Section 5.1.3).

6.3.3.2.2. Using Sender-Constrained Tokens

   Browser-based OAuth 2.0 clients can implement [DPoP] to transition
   from bearer access tokens and bearer refresh tokens to sender-
   constrained tokens. In such an implementation, the private key used
   to sign DPoP proofs is handled by the browser (a non-extractable
   CryptoKeyPair (https://developer.mozilla.org/en-US/docs/Web/API/
   CryptoKeyPair) is stored using IndexedDB). As a result, the use of
   DPoP effectively prevents scenarios where the attacker exfiltrates
   the application's tokens (See Section 5.1.1 and Section 5.1.2).

   Note that the use of DPoP does not prevent the attacker from running
   a new flow to obtain a fresh set of tokens (See Section 5.1.3). Even
   when DPoP is mandatory, the attacker can bind the fresh set of tokens
   to a key pair under their control, allowing them to calculate the
   necessary DPoP proofs to use the tokens.

6.3.3.2.3. Restricting Access to the Authorization Server

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

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

The authorization server could block authorization requests that
originate from within an iframe. While this would prevent the exact
scenario from Section 5.1.3, it would not work for slight variations
of the attack scenario. For example, the attacker can launch the
silent flow in a popup window, or a pop-under window. Additionally,
browser-only OAuth 2.0 clients typically rely on a hidden iframe-
based flow to bootstrap the user's authentication state, so this
approach would significantly impact the user experience.

The authorization server could opt to make user consent mandatory in
every Authorization Code flow (as described in Section 10.2 OAuth 2.0
[RFC6749]), thus requiring user interaction before issuing an
authorization code. This approach would make it harder for an
attacker to run a silent flow to obtain a fresh set of tokens.
However, it also significantly impacts the user experience by
continuously requiring consent. As a result, this approach would
result in "consent fatigue", which makes it likely that the user will
blindly approve the consent, even when it is associated with a flow
that was initialized by the attacker.

6.3.3.3. Summary

To summarize, the architecture of a browser-based OAuth 2.0 client
application is straightforward, but results in a significant increase
in the attack surface of the application. The attacker is not only
able to hijack the client, but also to extract a full-featured set of
tokens from the browser-based application.

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

7. Discouraged and Deprecated Architecture Patterns

Client applications and backend applications have evolved
significantly over the last two decades, along with threats, attacker
models, and our understanding of modern application security. As a
result, previous recommendations are often no longer recommended and
proposed solutions often fall short of meeting the expected security
requirements.

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

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

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

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

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

   * 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 multi-factor 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 above.

7.1.1. Threat Analysis

   Due to the lack of using OAuth, this architecture pattern is only
   vulnerable to the following attack payload: Proxying Requests via the
   User's Browser Section 5.1.4. As a result, this pattern can lead to
   the following consequence: Client Hijacking Section 5.2.3

7.2. OAuth Implicit 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 an
   authorization code exchange step. In this case, the access token is
   returned in the fragment part of the redirect URI, providing an
   attacker with several opportunities to intercept and steal the access
   token.

   Authorization servers MUST NOT issue access tokens in the
   authorization response, and MUST issue access tokens only from the
   token endpoint. Browser-based clients MUST use the Authorization
   Code flow and MUST NOT use the Implicit flow to obtain access tokens.

7.2.1. 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 OAuth 2.0
   Authorization Code flow with PKCE, since they have the ability to
   remove the authorization code from the query string without
   triggering a page reload thanks to the Session History API, and CORS
   support at the token endpoint means the app can obtain tokens even if
   the authorization server is on a different domain.

7.2.2. Threat Analysis

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

   * Single-Execution Token Theft Section 5.1.1

   * Persistent Token Theft Section 5.1.2

   * Acquisition and Extraction of New Tokens Section 5.1.3

   * Proxying Requests via the User's Browser Section 5.1.4

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

   * Exploiting Stolen Refresh Tokens Section 5.2.1

   * Exploiting Stolen Access Tokens Section 5.2.2

   * Client Hijacking Section 5.2.3

7.2.3. Further Attacks on the Implicit Flow

   Apart from the attack payloads and consequences that were already
   discussed, there are a few additional attacks that further support
   the deprecation of 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.

7.2.3.1. 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].

7.2.3.2. 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].

7.2.3.3. 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 attacks on the TLS layer 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.

7.2.3.4. 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.

7.2.4. 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. 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's token endpoint. Additionally, in many cases an application will request both an ID token and an access token, so it is simpler and provides fewer attack vectors to obtain both via the Authorization Code flow.

7.3. Resource Owner Password Grant

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 secure non-phishable authentication options, take advantage of single sign-on sessions, or use third-party identity providers. In contrast, the Resource Owner Password Credentials Grant does not provide any built-in mechanism for these, and would instead need to be extended with custom protocols.

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

7.4. Handling the OAuth Flow in a Service Worker

   In an attempt to limit the attacker's ability to extract existing
   tokens or acquire a new set of tokens, a pattern using a Service
   Worker (https://developer.mozilla.org/en-US/docs/Web/API/
   Service_Worker_API) has been suggested in the past. In this pattern,
   the application's first action upon loading is registering a Service
   Worker. The Service Worker becomes responsible for executing the
   Authorization Code flow to obtain tokens and to augment outgoing
   requests to the resource server with the proper access token.
   Additionally, the Service Worker blocks the client application's code
   from making direct calls to the authorization server's endpoints.
   This restrictions aims to target the attack payload "Acquisition and
   Extraction of New Tokens" (Section 5.1.3).

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

```
                                                          Resource
          Authorization
  User        Application          Service Worker            server
          server
    |   browse      |                    |                    |
    |               |                    |                    |
    | ------------->|                    |                    |
    |               |                    |                    |
    |               |------------------->                    |
/authorize    |     |                    |                    |
    |               |     ---------------------------------------------------
------------->|     |                    |                    |
    |               |                    |          redirect w/ authorization
 code          |    |                    |                    |
    |               |                    |  < - - - - - - - - - - - - - - - - - - -
  - - - - - |        |                    |                    |
    |               |                    |                    |
    |               |                    |                    |
    |               |                    | token request w/ auth code  |
    /token     |    |                    |                    |
    |               |                    | -----------------------------------------
------------->|     |                    |                    |
    |               |                    | <- - - - - - - - - - - - - - - - - - -
  - - - - - - -|     |                    |                    |
    |               |                    |                    |
    |               |                    |                    |
    |               | resource request  |                    |
    |               |                    |                    |
    |               |------------------->  resource request with token  |
    |               |                    |                    |
    |               |                    | --------------------------->|
    |               |                    |                    |
    |               |                    |                    |
    |               |                    |                    |
  User          Application        Service Worker            Resource
          Authorization
                                                             server
          server
```
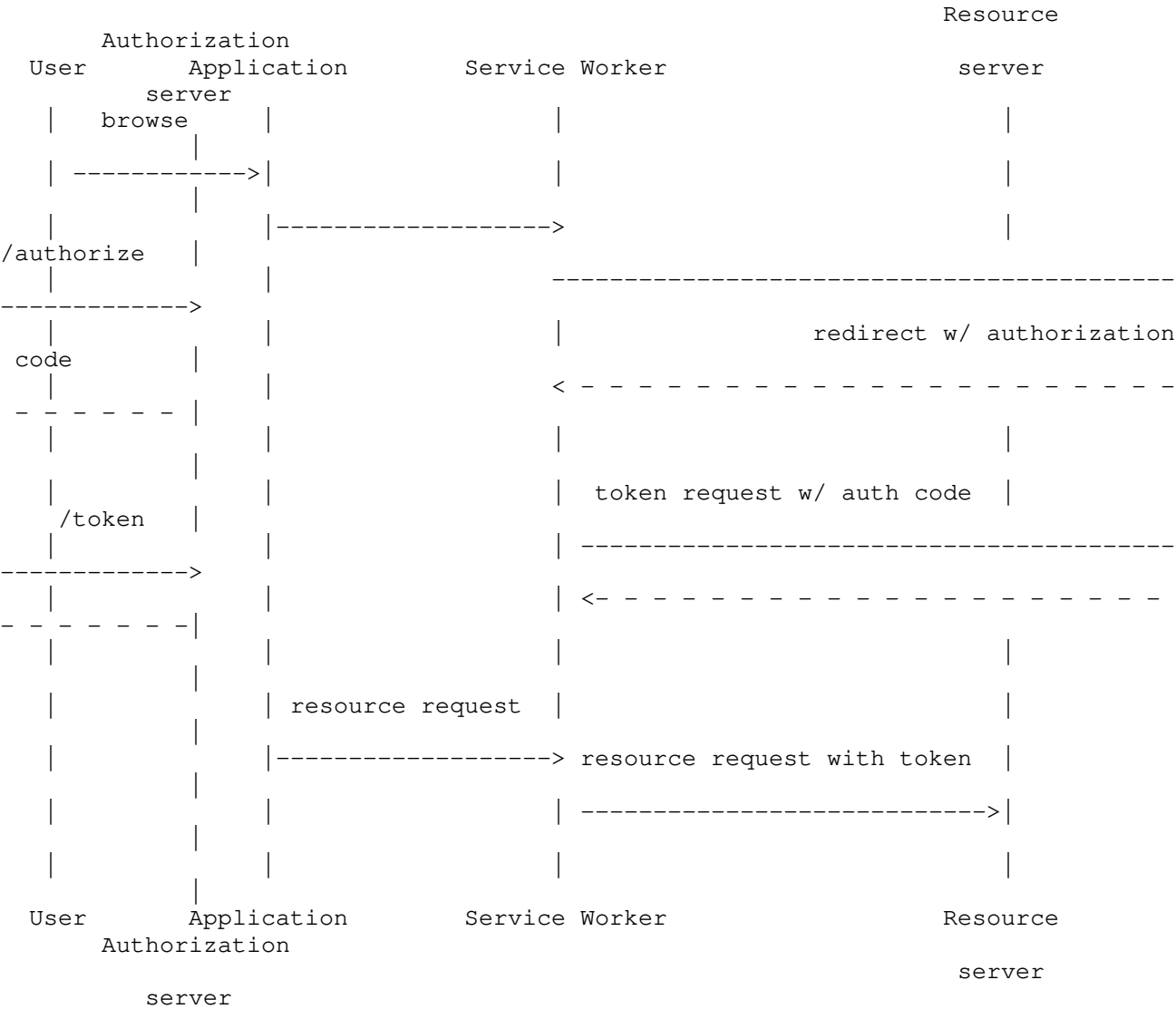
Note that this pattern never exposes the tokens to the application
running in the browser. Since the Service Worker runs in an isolated
execution environment, there is no shared memory and no way for the
client application to influence the execution of the Service Worker.

7.4.1. Threat Analysis

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

   * Acquisition and Extraction of New Tokens Section 5.1.3

   * Proxying Requests via the User's Browser Section 5.1.4

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

   * Exploiting Stolen Refresh Tokens Section 5.2.1

   * Exploiting Stolen Access Tokens Section 5.2.2

   * Client Hijacking Section 5.2.3

7.4.1.1. Attacking the Service Worker

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

   1. Prevent an attacker from exfiltrating tokens

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

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

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

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

One of the consequences of unregistering a Service Worker is that it
will not be present when a new browsing context is opened. So when
the attacker first unregisters the Service Worker, and then starts a
new flow in a frame, there will be no Service Worker associated with
the browsing context of the frame. Consequentially, the attacker
will be able to run an Authorization Code flow, extract the code from
the frame's URL, and exchange it for tokens.

In essence, the Service Worker fails to meet the second security
requirement, leaving it vulnerable to the payload where the attacker
acquires a new set of tokens (Section 5.1.3).

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

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

8. 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. Depending on the
   application's architecture, the tokens can include an access token
   and refresh token. Given the sensitive nature of refresh tokens, the
   application can decide to use different storage strategies for both
   types.

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

1. The attacker obtaining tokens from storage

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

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

That said, the different security properties of browser-based storage solutions will impact the attacker's ability to obtain existing tokens from storage. This section discusses a few different storage mechanisms and their properties.

8.1. Cookies

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

Next to header-based control over cookies, browsers also offer a JavaScript Cookie API to get and set cookies. This Cookie API is often mistaken as an easy way to store data in the browser. In such a scenario, the JavaScript code stores a token in a cookie, with the intent to retrieve the token for later for inclusion in the Authorization header of an API call. However, since the cookie is associated with the domain of the browser-based application, the browser will also send the cookie containing the token when making a request to the server running on this domain. One example of such a request is the browser loading the application after a previous visit to the application (step A in the diagram of Section 6.3).

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

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

8.2. Token Storage in a Service Worker

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

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

As discussed before, the use of a Service Worker does not prevent an attacker from obtaining a new set of tokens. Similarly, if the Service Worker initially obtains the tokens from the legitimate application, the attacker can likely obtain them in the same manner.

8.3. Token Storage in a Web Worker

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

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

One common use of Web Workers is to isolate the refresh token. In such a scenario, the application runs an Authorization Code flow to obtain the authorization code. This code is forwarded to a Web Worker, which exchanges it for tokens. The Web Worker keeps the

refresh token in memory and sends the access token to the main
application. The main application uses the access token as desired.
When the application needs to run a refresh token flow, it asks the
Web Worker to do so, after which the application obtains a fresh
access token.

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

8.4. In-Memory Token Storage

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

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

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

8.5. Persistent Token Storage

The persistent storage APIs currently available 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 the lifetime of sessionStorage is linked to the lifetime of a browser tab. Additionally, sessionStorage is not shared between multiple tabs open to pages on the same origin, which slightly reduces the exposure of the tokens in sessionStorage.

IndexedDB is a persistent storage mechanism like localStorage, but is shared between multiple tabs as well as between the browsing context and Service Workers.

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

8.6. Filesystem Considerations for Browser Storage APIs

In all cases, as of this writing, browsers ultimately store data in plain text on the filesystem. This behavior exposes tokens to attackers with the ability to read files on disk. While such attacks rely on capabilities that are well beyond the scope of browser-based applications, this topic highlights an important attack vector against modern applications. More and more malware is specifically created to crawl user's machines looking for browser profiles to obtain high-value tokens and sessions, resulting in account takeover attacks.

While the browser-based application is incapable of mitigating such attacks, the application can mitigate the consequences of such an attack by ensuring data confidentiality using encryption. The [WebCryptographyAPI] provides a mechanism for JavaScript code to generate a secret key, as well as an option for that key to be non-exportable. A JavaScript application could then use this API to encrypt and decrypt tokens before storing them. However, the [WebCryptographyAPI] 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. Security Considerations

9.1. Reducing the Authority of Tokens

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

   * Reduce the lifetime of access tokens and rely on refresh tokens
      for straightforward access token renewal

   * Reduce the scopes or permissions associated with the access token

   * Use [RFC8707] to restrict access tokens to a single resource

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

9.2. Sender-Constrained Tokens

   As discussed throughout this document, the use of sender-constrained
   tokens does not solve the security limitations of browser-only OAuth
   clients. However, when the level of security offered by a token-
   mediating backend (Section 6.2) or a browser-only OAuth client
   (Section 6.3) suffices for the use case at hand, sender-constrained
   tokens can be used to enhance the security of both access tokens and
   refresh tokens. One method of implementing sender-constrained tokens
   in a way that is usable from browser-based apps is [DPoP].

   When using sender-constrained tokens, the OAuth client has to prove
   possession of a private key in order to use the token, such that the
   token isn't usable by itself. If a sender-constrained token is
   stolen, the attacker wouldn't be able to use the token directly, they
   would need to also steal the private key. In essence, one could say
   that using sender-constrained tokens shifts the challenge of securely
   storing the token to securely storing the private key.

   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 [WebCryptographyAPI].
   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 its own execution context. The techniques for doing so are similar to using a secure token storage mechanism, as discussed in Section 8.

While a non-exportable key is protected from exfiltration from within JavaScript, exfiltration of the underlying private key from the filesystem is still a concern. As of the time of this writing, there is no guarantee made by the [WebCryptographyAPI] that a non-exportable key is actually protected by a Trusted Platform Module (TPM) or stored in an encrypted form on disk. Exfiltration of the non-exportable key from the underlying filesystem may still be possible if the attacker can get access to the filesystem of the user's machine, for example via malware.

9.3. Authorization Server Mix-Up Mitigation

Authorization server mix-up attacks mark a severe threat to every client that supports at least two authorization servers. 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 [RFC9207], or by using the iss claim of the ID token when using OpenID Connect.

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.

9.4. Isolating Applications using Origins

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

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

10. IANA Considerations

   This document does not require any IANA actions.

11. References

11.1. Normative References

   [CookiePrefixes]
           Contributors, M., "Using HTTP cookies", n.d.,
           <https://developer.mozilla.org/en-US/docs/Web/HTTP/
           Cookies>.

   [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-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>.

   [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated
           Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008,
           <https://www.rfc-editor.org/info/rfc5116>.

   [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>.

   [RFC9207] Meyer zu Selhausen, K. and D. Fett, "OAuth 2.0
              Authorization Server Issuer Identification", RFC 9207,
              DOI 10.17487/RFC9207, March 2022,
              <https://www.rfc-editor.org/info/rfc9207>.

11.2. Informative References

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

   [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>.

   [RFC8707] Campbell, B., Bradley, J., and H. Tschofenig, "Resource
              Indicators for OAuth 2.0", RFC 8707, DOI 10.17487/RFC8707,
              February 2020, <https://www.rfc-editor.org/info/rfc8707>.

   [Site]   Contributors, M., "Site", n.d.,
              <https://developer.mozilla.org/en-US/docs/Glossary/Site>.

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

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

   [WebMessaging]
              whatwg, "HTML Living Standard - Cross-document messaging",
              December 2023, <https://html.spec.whatwg.org/multipage/
              web-messaging.html#web-messaging>.

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 6.3.2.1.1

   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 for browser-based clients.

   5. Require exact matching of registered redirect URIs for browser-
       based clients.

   6. Support cross-domain requests at endpoints browser-based clients
       access in order to allow browsers to make the authorization code
       exchange request. See Section 6.1.3.3.2

   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 6.3.2.7.

Appendix B. Document History

   [[To be removed from the final specification]]

   -17

* Added a section on anti-forgery/double-submit cookies as another
  form of CSRF protection

* Updated CORS terminology

* Moved new section on in-browser flows as not applicable to BFF or
  TM patterns

* Fixed usage of some browser technology terminology

* Editorial improvements

-16

* Applied editorial changes from Filip Skokan and Louis Jannett

* Clarified when cookie encryption applies

* Added a section with security considerations on the use of
  postMessage

-15

* Consolidated guidelines for public JS clients in a single section

* Added more focus on best practices at the start of the document

* Restructured document to have top-level recommended and
  discouraged architecture patterns

* Added Philippe De Ryck as an author

-14

* Minor editorial fixes and clarifications

* Updated some references

* Added a paragraph noting the possible exfiltration of a non-
  exportable key from the filesystem

-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,
Damien Bowden, Daniel Fett, Elar Lang, Eva Sarafianou, Filip Skokan,
George Fletcher, Hannes Tschofenig, Janak Amarasena, John Bradley,
Joseph Heenan, Justin Richer, Karl McGuinness, Karsten Meyer zu
Selhausen, Leo Tohill, Louis Jannett, Mike Jones, 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

Philippe De Ryck
Pragmatic Web Security
Email: philippe@pragmaticwebsecurity.com