ORACLE

Oracle Cloud Infrastructure IAM Identity Domain OAuth and OpenID Connect Flows and Best Practices

February 2023, version 1.0

Disclaimer

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle software license and service agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

This document is for informational purposes only and is intended solely to assist you in planning for the implementation and upgrade of the product features described. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle. Due to the nature of the product architecture, it may not be possible to safely include all features described in this document without risking significant destabilization of the code.

Revision History

The following revisions have been made to this document.

DATE	REVISION
February 2023	Initial publication

Table of Contents

Overview **4**

Terminology **4**

OAuth 2.0 Authorization and OpenID Connect Framework **6**

OAuth 6

OpenID Connect 6

OAuth Flows **7**

Client Credential Flow 7

Authorization Code Flow 8

Authorization Code with PKCE Flow 10

Refresh Token Flow 11

Device Authorization Flow 11

Assertion Framework 11

TLS Client Authentication Flow 12

Implicit Flow 13

Resource Owner Password Credential Flow 13

Best Practices **13**

Which Flow to Use 13

Best Practices for Single-Page Apps 14

Best Practices for Native Mobile Apps 16

Best Practices for Token Validation 17

General Best Practices 17

Example Requests and Responses **18**

References **19**

Overview

Oracle Cloud Infrastructure Identity and Access Management (OCI IAM) identity domains support the OAuth 2.0 authorization framework drafted by the Internet Engineering Task Force (IETF) and the OpenID Connect core specification. Protecting an application with OAuth 2.0 and OpenID Connect (OAuth/OpenID Connect) requires the use of various specifications. This paper describes the various flows and best practices of OCI IAM identity domains with OAuth/OpenID Connect.

Terminology

The OAuth/OpenID Connect protocol provides several flows to authenticate and authorize your applications. This section defines various terms used in OAuth/OpenID Connect flows.

- **Resource:** The protected data owned by an entity. In a typical use case, a resource is the data owned by a person—for example, your physical business card, digital profile (name, email, and so on), photos, and contacts.
- **Resource owner:** An entity that owns the protected resource. Typically, the resource owner is a person who owns the resource (data) and can grant access to this data to various applications. For example, you own your business card and can give it to someone when you want to. Similarly, you can grant access to your digital profile, photos, or contacts to some other application.
- **Resource server:** A system or server that holds the protected resource. The resource server can send this data back to the requesting application on behalf of a user who authorized such requests. For example, when you upload your photos to a cloud service, the cloud service is the resource server that is holding your photos (protected resource).
- **Client:** An application that wants to access your protected resource on your behalf with your permission. For example, a printing web application that can print your photos and then mail them to you might want to access the photos from the cloud service on your behalf, so that you don't have to upload them again to the application.
- **Authorization server**: A system that allows a *resource owner* (you) to authenticate themselves and then authorize a *client* (printing web application) access to a protected *resource* (your photos) on your behalf. OCI IAM is an authorization server.
- **Token:** A text string that represents arbitrary data or information.
- **Hashing:** Converting text or data into a fixed-character string by using an algorithm. For example, the hash value of the text "Hello" using the SHA256 algorithm is `185f8db32271fe25f561a6fc938b2e264306ec304eda518007d176482638196`. Hashing is one way only and always produces the same result. For example, you can't get the text "Hello" from its hash value, but you always get the same hash value for "Hello" when you use the same algorithm.
- **Signature**: A text string generated by converting data using some algorithm and the private key of the owner. The receiver can verify a signature by using the owner's public key. Tokens are digitally signed by the issuer of the token, and a client and resource server verify the signature of the token before using it.
- **Java Script Object Notification (JSON):** An easy-to-read-and-understand plain-text format for storing and transmitting data. The format consists of data in key/value pairs in which the key is a string and the value is a string or another JSON string. For example:

```
{"name":"John Doe", "age":"88", "address":"123 First Ave, Mars", "email":
{"work":"work@example.com", "home":"home@example.com"}}
```

- **JSON Web Token (JWT):** A compact string that represents a set of data in JSON format. JWT consists of three parts: a header, the payload (data), and a signature. These parts are base64UrlEncoded, which means converting a string to a format that can be safely transmitted over the internet without losing any characters. They're also separated by dots (`.`). For example:

  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJqb2huLmRvZSIsIm5hbWUiOiJKb2huIERvZSIsImlhdCI6MTUxNjIzOTAyMn0.5kY5qwLC0cBX4eSSuqkfABmWjrJ5kXmaE9wcGSMeNa4

  If we base64UrlDecode the preceding string, it looks as follows:

Header	Payload	Signature

{
"alg":"HS256",
"typ":"JWT"
}
The value HS256 defines the algorithm used to sign the payload.
	{
	"sub":"john.doe",
	"name":"John Doe",
	"iat":1516239022
	}
		5kY5qwLC0cBX4eSSuqkfABmWjrJ5kXmaE9wcGSMeNa4
		The signature is the signed value of the input *header*. *payload*, signed by the issuer or creator of the JWT.

  A JWT (or *token*) is represented by either JSON Web Signature (JWS) or JSON Web Encryption (JWE) standards. These standards define the token format and how tokens are digitally signed and encrypted. Most commonly, tokens are represented in JWS, which allows anyone to read the content of the token. Tokens represented in JWE are encrypted. Although you can read tokens represented in JWS, they're still secure because they're digitally signed by the creator and the consumer can verify their signature.

- **Identity provider (IdP):** An OpenID Connect provider, such as OCI IAM, that authenticates users and issues ID tokens.
- **Scope:** Information in the form of a string requested by a client from the authorization server. For example, *scope=email*. When a client requests a scope, the request isn't guaranteed to be fulfilled; the end user can deny the request.
- **Claim:** Information in a key/value pair, typically about a user, asserted by the IdP. Example: `"email":"john.doe@example.com"`
- **ID token**: A JWT token, issued by an IdP to a client. An ID token identifies who a user is, and its payload contains claims such as subject, name, and issued at (iat). A client can use an ID token to display a user profile in the application. ID tokens are meant to be read by the client to identify a user's identity.
- **Access token:** A JWT token. As the name suggests, an access token allows the client to access a resource. The authorization server grants an access token to a client with the appropriate scope (what a client can access) and the client uses it to access the resource. Access tokens are also called *bearer tokens*, which means that anyone in possession of the token can use it to access the resource.

OAuth 2.0 Authorization and OpenID Connect Framework

OCI IAM Identity domains support both the OAuth 2.0 and the OpenID Connect frameworks. This section briefly describes these frameworks.

OAuth

Before OAuth, if an application needed to access your data (for example, your contacts), then it asked for your credentials (email and password) to fetch the data. For example, social networking applications asked for your email and password to find out if your contacts were also in the same social networking application so that it could automatically send friend requests for you. Although this process was easy to use, it was highly unsecure. For example, you might trust some applications with your email passwords, but what about applications you don't trust? What if applications store your email/password for future use, or write them in the logs? How do you track what applications you gave your email and password to and how do you remove that information from all the applications? The OAuth framework overcomes all these issues and more.

Let's consider the same example: a social networking application wants to access your contacts. With OAuth, the application doesn't ask for your email and password; instead, it takes you to your email service provider to authenticate yourself and then obtain consent from you to access the contacts. After that, the application gets limited access to access your contacts. Later, you can remove the consent given to the application.

The OAuth framework enables a client to obtain limited access to a resource on behalf of the resource owner or on its own behalf. It allows the resource owner to give consent on the information to be shared with the application. OAuth separates the authentication of the client (that is, an application) and the resource owner. Instead of using the resource owner username and password to access the resource, owned by the owner, the client gets an access token with limited scope (what information the client can access), expiry, and other access attributes, and uses it to access the resource.

OpenID Connect

OAuth allows clients to access data on behalf of a user upon their consent. Although OAuth handles the authorization, it doesn't mandate verification of the user's identity. Before OpenID Connect was introduced, the client didn't have a standard way to verify a user's identity.

OpenID Connect 1.0 is an identity layer on top of OAuth 2.0. It lets clients verify the identity of a user based on the authentication done by an authorization server. It also lets clients obtain basic profile information about the user in an interoperable and REST-like manner.

OpenID Connect uses the OAuth flows. When the client adds the `openid` scope to the OAuth request, the authorization server returns an ID token, an access token, and a refresh token (optional). The ID token verifies the user's identity to the client. After receiving the ID token, the client must validate the token and can then create a session. In a later section, we look at when and how ID tokens are returned in various flows and best practices for validating the ID token.

OAuth Flows

The OAuth framework defines several roles, such as resource owner, resource server, client, and authorization server. It also defines several flows (called *grant types*): client credential, authorization code, refresh token, device authorization, assertion, transport layer security (TLS) client authentication, implicit, and resource owner password credential.

This section describes these flows. The following table describes the various endpoints that the OAuth flows and OCI IAM use.

OCI IAM Identity Domain OAuth Endpoints

OCI IAM OAUTH ENDPOINTS	URL	DESCRIPTION
Token	`<domain-url>/oauth2/v1/token`	Generate an access token, ID token, or refresh token to access the resource. Clients call this endpoint from within the application rather than in the browser.
Authorization	`<domain-url>/oauth2/v1/authorize`	Generate an authorization code that clients exchange for tokens.
Logout	`<domain-url>/oauth2/v1/userlogout`	Log out a user from the OCI IAM identity domain and remove associated cookies and sessions.
Introspect	`<domain-url>/oauth2/v1/introspect`	Obtain information about the circumstances under which a token was created. Data returned includes the expiry date of the token, the intended audience, and any assurance level that was associated. This endpoint delivers context about the user's current session.
Revoke refresh token	`<domain-url>/oauth2/v1/revoke`	Revoke a refresh token.
User info	`<domain-url>/oauth2/v1/userinfo`	Generate user details using an access token.
Discovery	`<domain-URL/oauth2/v1/.well-known/openid-configuration`	Retrieve OpenID discovery documentation.
Consents	`<domain-url>/admin/v1/OAuthConsents`	Manage OAuth consents.

Client Credential Flow

In the OAuth client credential flow, the client is typically a server, a CLI, or a desktop application; user authentication isn't involved. This flow has the following main steps:

1. The client authenticates with the authorization server's token endpoint and gets an access token in return.
2. The client uses the access token to invoke the API and gets the protected data in return.

The following diagram illustrates this flow in detail with OCI IAM. Before the flow begins, the client application registers itself with the OCI IAM authorization server. As part of the registration, the client chooses the flow (grant type) as the client credential and chooses the app role or roles that it needs to access the APIs. For example, if the client needs to call the Create User API, the client needs the User Administrator app role.

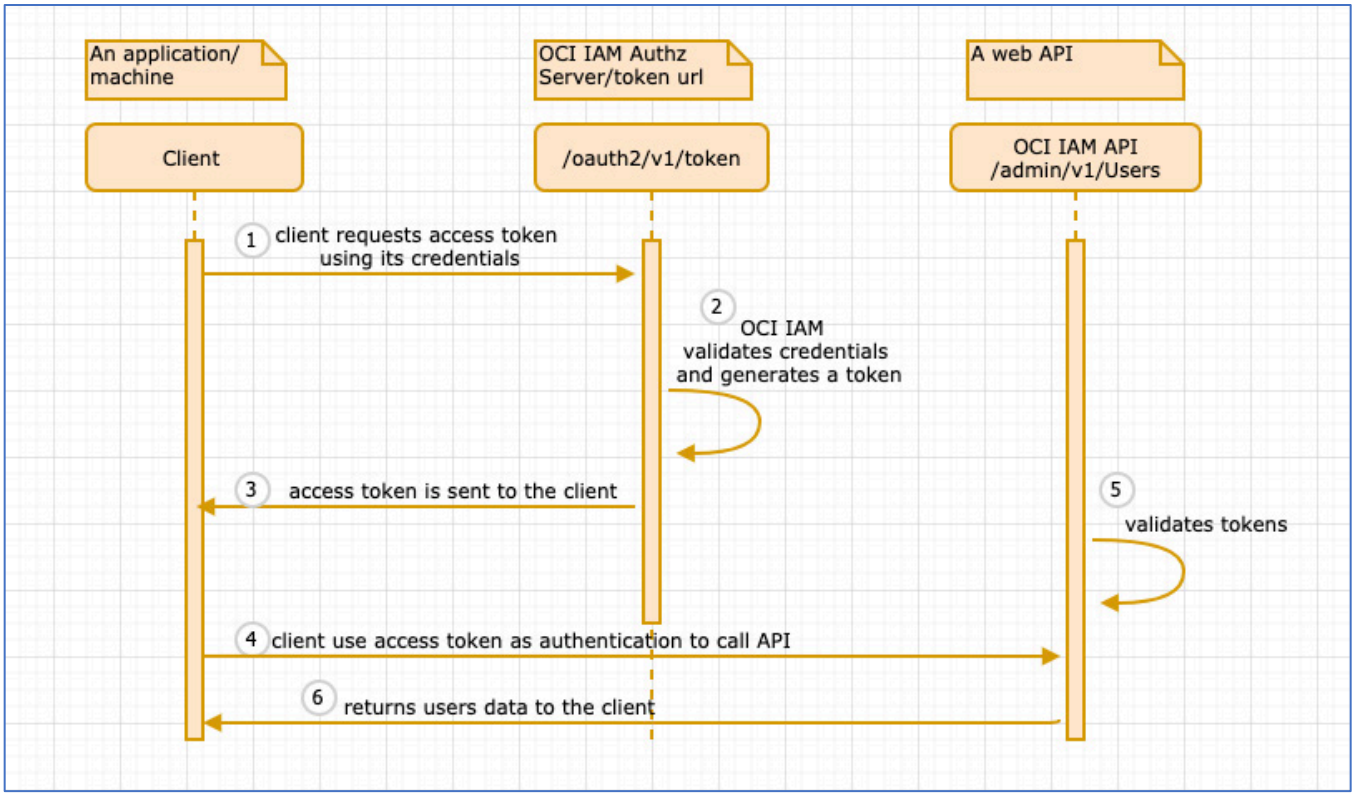

Figure 1: Client Credential Flow

The following steps are illustrated in the diagram:

1. The client application creates a request and submits it to obtain an access token from the OCI IAM identity domain OAuth2.0 authorization server's token endpoint (/oauth2/v1/token).
2. The OCI IAM authorization server validates the credentials and generates an access token.
3. The authorization server sends an access token in the response. If authorization wasn't successful, it sends an error message instead.
4. The client creates another request to call an API on behalf of itself and passes the access token in the authorization header for authentication.
5. The OCI IAM identity domain API verifies that this access token has access to call the requested API.
6. If successful, the API sends back the user data in the response. Otherwise, it sends an error message.

Authorization Code Flow

In the authorization code flow, the client is typically a web application that involves user authentication. This flow has the following main steps:

1. The client submits an authorization request through a browser to the authorization server's authorization endpoint. The authorization server authenticates the user and asks the user to approve or deny consent for the client to access the user's resource.
2. In return, the client gets an authorization code that it exchanges for an access token at the token endpoint. The client uses the access token to invoke the API and gets the protected data in return.

The following diagram illustrates this flow in detail with OCI IAM. Before the flow begins, the client application first registers itself with the OCI IAM authorization server.

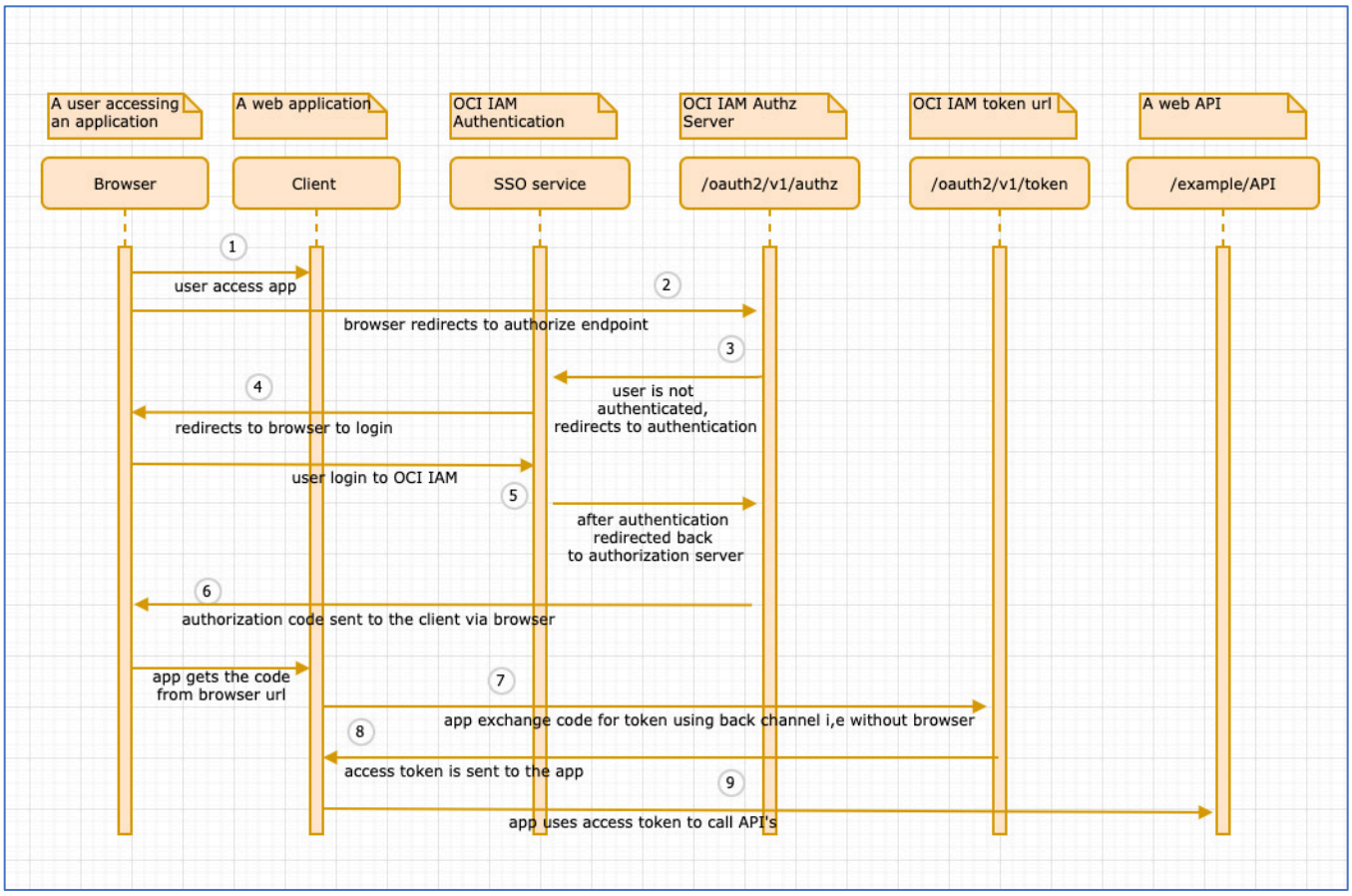

Figure 2: Authorization Code Flow

The following steps are illustrated in the diagram:

1. The user accesses the application by using a browser, and the application creates an authorization request with scope.
2. The browser redirects the application to the authorization endpoint.
3. If the user isn't authenticated, the authorization server redirects to the authentication service for login.
4. The user logs in and OCI IAM validates the credentials.
5. After authentication, the user is redirected back to authorization server with a request. The authorization server sends a consent page to the browser that lists the requested scope, and the user allows or denies consent. If denied, an error is shown and the flow isn't completed.
6. The authorization server creates a one-time authorization code and sends it to the client through the browser.
7. The application gets the code from the browser URL and exchanges the code with the token endpoint by directly calling the token endpoint.
8. The authorization server sends an access token, an ID token (if the `openid` scope is requested in step 1), and a refresh token (if the `offline_access` scope is requested in step 1) to the client.
9. The client application uses the access token to call the API.

Authorization Code with PKCE Flow

In the authorization code flow with Proof Key for Code Exchange (PKCE), the client is typically a single-page app or a mobile app that involves user authentication. Such applications can't securely store client secrets in the application (public clients).

The following diagram illustrates this flow in detail with OCI IAM. Before the flow begins, the client application first registers itself with the OCI IAM authorization server.

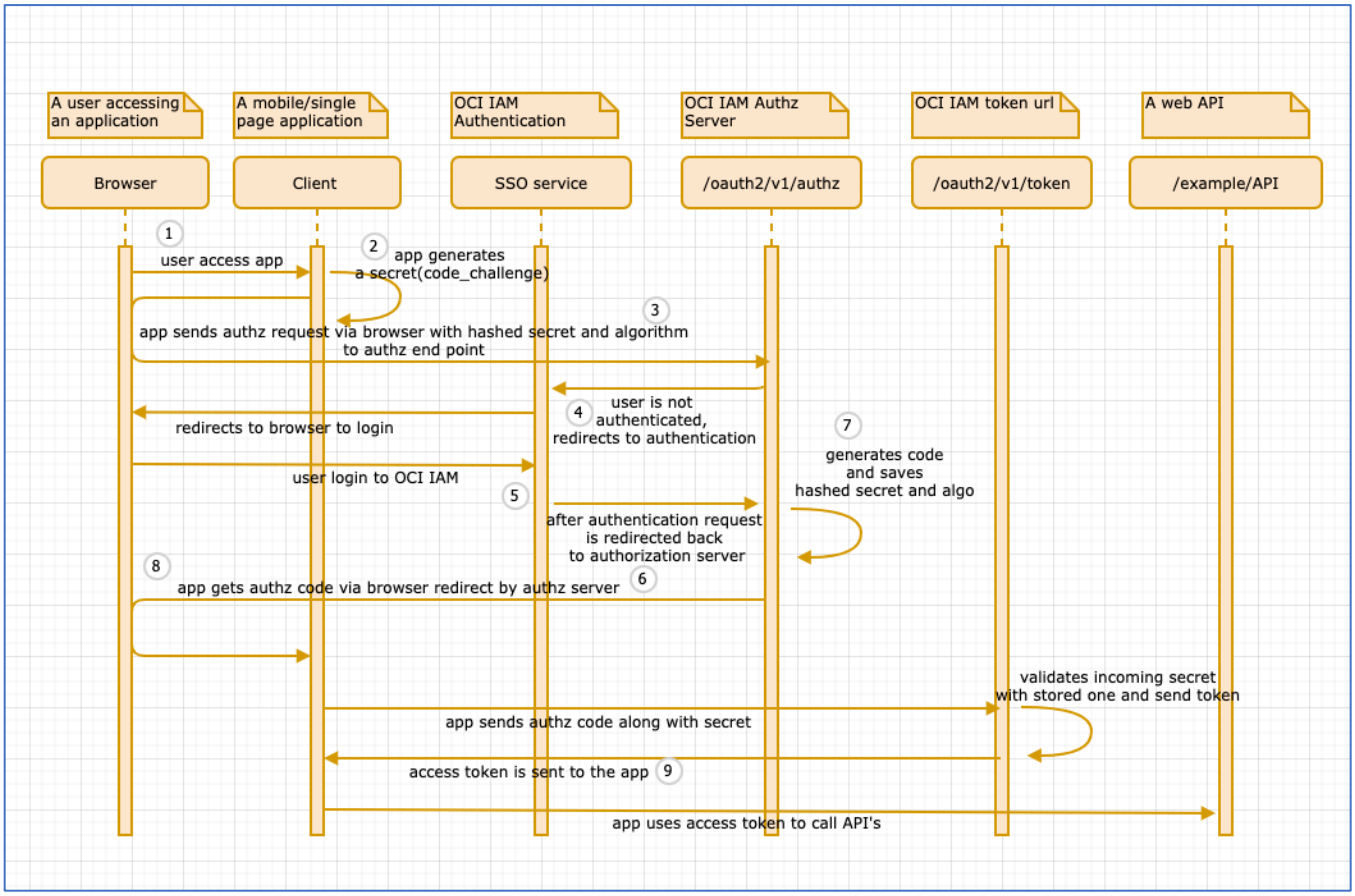

Figure 3: Authorization Code with PKCE Flow

The following steps are illustrated in the diagram:

1. The user accesses the app by using a browser.
2. The app creates a secret (`code_verifier`), saves it in a persistent store, and creates an authorization request.
3. The app sends the authorization request with a hashed secret (`code_challenge`) and the algorithm used for hashing (`code_challenge_method`) to the authorization server.
4. If the user isn't authenticated, the authorization server redirects to authentication service for login.
5. The user logs in and OCI IAM validates the credentials.
6. After authentication, the user is redirected back to authorization server with a request. The authorization server sends a consent page to the browser that lists the requested scope, and the user allows or denies consent. If denied, an error is shown and the flow isn't completed.
7. Authorization server saves the hashed secret and algorithm, creates a one-time authorization code, and sends it to the client through the browser.

8. The app gets the code from the browser URL, exchanges the code with token endpoint, and passes the stored secret.
9. The authorization server calculates the hash of the incoming secret. If it matches with stored hash, an access token, an ID token (if the `openid` scope is requested in step 3), and a refresh token (if the `offline_access` scope is requested in step 3) is sent to the client.

Refresh Token Flow

When a client needs to access a resource on behalf of a user, the client uses flows such as the authorization code flow to get an access token. As part of flow, the user must authenticate themselves. Making the user log in every time might not be optimal behavior for some applications. As an alternative, the client can use the refresh token flow to silently get the access token by using a refresh token that the client obtains as part of the initial request.

Access tokens are short-lived, and when they expire, the client must run the authentication flow again to get the access token. However, with the refresh token flow, the client can send a request to the token endpoint by passing an existing refresh token to get a new access token/refresh token pair.

As an extra security measure, OCI IAM creates a new refresh token with each exchange of the existing refresh token. As a result, the client must use the new refresh token to obtain an access token using refresh token flow.

Device Authorization Flow

Many “smart” devices, such as smart TVs and media consoles, don’t have a browser, or it’s difficult to get the user input to complete the authorization code flow. The device authorization flow uses the external agents from a device so that a user can easily access the browser. This flow happens in two places:

- **On the device**: The device requests a device code, user code, and verification URL from the authorization server device endpoint. The device then starts polling the authorization server for tokens.
- **On the user mobile or desktop**: The user accesses the verification URL from a browser, enters the user code noted from the device, authenticates, and consents, which completes the flow. The device gets the token and can then call the resource server API by using the token.

Assertion Framework

In OAuth flows such as the authorization code flow, the client first gets authorization from the authorization server on behalf of a user and then the user consents before giving the authorization to the client. After the client gets the authorization code, it exchanges it for a token at the token endpoint and uses the token to call APIs.

The OAuth Assertion specification provides a framework for using assertion as an authorization grant. Assertion can bridge the gap between OAuth and other protocols, and be used in integration scenarios in which user interaction isn’t involved. Assertion can also be used as a client authentication.

Two OAuth profiles, SAML 2.0 for OAuth and JWT for OAuth, further discuss the assertion framework. The following sections summarize how JWTs can be used as authorization grants and for client authentication. Assertion is signed by the key cryptography in which the private key never leaves the device or application.

JWTs as Authorization Grants

When using a JWT as an authorization grant, the flow is as follows:

1. The administrator uploads the public key that corresponds to the private key used to sign the assertion to the authorization server. The public key is used by the authorization server to validate the JWT signature.
2. The client creates a JWT assertion, which is a JWT with a header, payload, and signature.
3. The client creates a request to the token endpoint with the grant type as `urn:ietf:params:oauth:grant-type:jwt-bearer` and passes the assertion in the form of a single JWT that represents a user.
4. The authorization server validates the incoming JWT for claims such as subject, issuer, audience, and expiry, and verifies that it trusts the issuer of the JWT before issuing an access token.

This flow doesn't return a refresh token. When the access token expires, the client can submit a new request, with the same JWT if it isn't expired or a new one, to get the access token.

JWTs for Client Authentication

In OAuth flows, to prove its identity, a client must authenticate itself to the various OAuth endpoints such as token, refresh, and introspection. Typically, a client authenticates by using a shared secret (`client_secret`) that is known to the client and the authorization server. This secret is passed as part of the token request and is acceptable in most typical scenarios. However, a sensitive application might require that a client *not* use a shared secret or pass the secret with the token request to get the token. Essentially, the credential never leaves the device or application. For such use cases, the client can authenticate by using a JWT assertion. The flow is as follows:

1. The administrator uploads client's public key that corresponds to private key used to sign the client assertion to the authorization server. The public key is used to validate the JWT signature.
2. The client creates a JWT client assertion, which is a JWT with the required header, payload, and signature. The signature is created by signing the hash of the header and payload with the client's private key.
3. The client sends the JWT assertion with a token request.
4. After receiving the request, the authorization server validates the incoming token and then validates the signature. If it matches, the authorization server successfully authenticates the client.

TLS Client Authentication Flow

The TLS client authentication flow uses mutual transport layer security (mTLS) communication between the client and OCI IAM. In a mTLS connection, the client validates the server identity, and the server requests that that client prove its identity. OCI IAM has a separate mTLS endpoint, and clients should submit all requests to this endpoint when using this flow.

1. The client establishes a mTLS connection with the OCI IAM mTLS endpoint, and as part of the handshake, presents its client ID and certificate.
2. When a successful mTLS connection is established, the authorization server authenticates the client by using its certificate. The server then returns an access token to the client with claims according to the application role assigned to the client application.

Implicit Flow

In the implicit flow, an access token is returned directly in the redirect URL as part of the authorization response. Contrast this with the authorization code flow, in which an authorization code is obtained from the authorization endpoint and is later exchanged for the access token from the token endpoint. Implicit flow is vulnerable to access token leakage and replay attack, and best practice is to use the authorization code flow with PKCE instead of the implicit flow. Implicit flow is also not included in the OAuth 2.1 draft.

If you have an application that's using the implicit flow, evaluate the current security posture of the application and consider upgrading it to a more secure OAuth flow such as authorization code with PKCE.

Resource Owner Password Credential Flow

The resource owner password credential (ROPC) flow requires user credentials to be shared with the client. The client posts a request to the authorization server, and the server returns the access token immediately. Unlike other browser-based flows in which users are redirected to an IdP to authenticate and the IdP can further secure authentication by using multifactor authentication, this flow can't use IdP-enabled authentication. Because users have to share credentials with the client app, best practice is to *not* use the ROPC flow. The ROPC flow is also not included in the OAuth 2.1 draft.

If you have an application that's using this flow, evaluate the current security posture of the application and consider upgrading it to a more secure OAuth flow such as authorization code with PKCE.

Best Practices

This section describes best practices for using OAuth/OpenID Connect in the most widely used flows. *We recommend that you monitor security proactively so that you can quickly identify new security threats.*

Which Flow to Use

The following tables highlight some points to consider when choosing an OAuth flow. Because best practice is to *not* use the implicit or ROPC flow for newer applications, they're not included in the tables.

CLIENT CREDENTIAL	AUTHORIZATION CODE	AUTHORIZATION CODE WITH PKCE
• Use this flow for server-to-server authorization, in which the *client* is also the *resource owner*. That is, the client doesn't need to know the user credentials, or user authorization isn't required. • Use this flow when you can securely store a *client secret* in the application backend. • Because no user is involved, the access token is issued in the context of the client.	• Use this flow for a typical web application that must access a resource on a user's behalf, and user authorization is required. • Use this flow when you *can* securely store a *client secret* in the application backend that doesn't run code in the browser.	• Use this flow for native mobile apps, hybrid mobile apps, and single-page web apps that must access a resource on a user's behalf, and user authorization is required. • Use this flow when you *can't* securely store a *client secret* in the application.
• A refresh token isn't issued because user login isn't involved, and the client can rerun the flow to get a new token when the access token expires. • An example of this flow is a test-to-production application or script moving users and groups from one OCI IAM Identity domain to another.	• Use this flow when you want to leverage an external IdP. OCI IAM supports external IdPs for authentication, so when the flow prompts for user authentication, the user can choose to authenticate with the external IdP. • Use this flow when you don't want users to have to log in frequently. You can use long-lived refresh tokens to get new access tokens. • Because the client requests authorization on behalf of the user, the access token is issued in the context of the end user.	

DEVICE FLOW	ASSERTION FLOW	TLS CLIENT AUTHENTICATION
• Use this flow for devices that don't have a browser or are constrained in the way that they get user input (for example, a smart TV). • The device can use a refresh token to get a new access token so that frequent user login isn't needed.	• User assertion: Use this flow to leverage an existing trust relationship and choose when the party asserting the JWT is trustworthy. • Client assertion: Use this flow for sensitive applications that have stricter compliance requirements for client authentication.	• This flow uses key cryptography for confidential clients only that can securely store a private key. • Use this flow when regulatory compliance requirements mandate mTLS authentication and don't approve shared secrets.

Best Practices for Single-Page Apps

Single-page apps are web applications written in JavaScript that runs on the browser. Typically, single-page apps don't have any backend components and the application code is available and run in the browser. These apps call APIs by using JavaScript to display data in the UI without loading the whole page. Single-page apps are widely used, and there are certain security challenges when using OAuth with them. This section discusses those challenges and some best practices.

Security Challenges

A single-page app can't store client secrets or sensitive information such as access tokens or refresh tokens securely because browser-based storage mechanisms (for example, local storage) are unsafe and susceptible to attacks. Any malicious JavaScript code can read the sensitive information stored in the local storage or read an access token from the browser. Because single-page apps can't securely store client secrets, they're considered public clients for security best practices.

Implicit Flow and Authorization Code Flow

The preceding challenges make the use of implicit flow less secure for single-page apps. In implicit flow, access tokens are returned in the browser URL from the authorization server and are susceptible to attacks. The next secure OAuth flow is the authorization code flow.

In the code flow, when the client exchanges a code for a token, the client also sends a client secret by directly calling the token endpoint. This client secret lets the authorization server know that it's the same client to whom it had issued the code. Therefore, if any bad actor or malicious code has read the authorization code from the browser, they can't use it because the authorization server validates the client secret.

However, the code flow can't be used with single-page apps because they're public clients and can't have client secrets. In fact, OCI IAM doesn't generate client secrets for public client apps for security reasons. Alternatively, the authorization code with PKCE flow (PKCE flow) prevents authorization code interception attacks and doesn't use client secrets. The PKCE flow is a best practice for single-page apps.

Authorization Code with PKCE Flow and Refresh Token Rotation

In the PKCE flow, when the app requests the authorization code from the authorization server, it also passes a hash of a secret known to the client along with the hashing algorithm used. The authorization server saves this information. Later, when the app exchanges the code for the token from the token endpoint, it passes the original challenge again. The authorization server calculates the hash of the challenge and validates it against the stored hash, and upon successful validation, issues the token. This flow prevents an authorization code interception attack in single-page apps. If the code is stolen and sent to the token endpoint, the attacker also needs the original challenge, which is known only to the app, and the access token isn't issued.

Other advantage of the PKCE flow is that it can return a refresh token, and the app can use the refresh token flow to get a new access token. But this advantage comes with a security challenge. Refresh tokens are typically long-lived, and if a malicious actor can get one, the actor can generate access tokens indefinitely by using it. To mitigate this risk to some extent, single-page apps can use refresh token rotation.

In refresh token rotation, the app requests a refresh token when the access token is expired. OCI IAM returns a new refresh token and invalidates the old refresh token. The app must use the new refresh token to get another access token/refresh token pair, and it can continue doing this whenever the access token is expired. OCI IAM also prevents abuse of refresh tokens. If OCI IAM detects that a request is trying to use a refresh token that was already used by another application, it flags the token as stolen and invalidates all the tokens.

In summary, by using the authorization code with PKCE flow and refresh token rotation, a single-page app can be secured to a great extent but not completely. An attacker or malicious code running inside the app can bypass the refresh token abuse detection by waiting for the app to be inactive and then using the refresh token to request a new access token.

Backend for Frontend Architecture

The backend for frontend (BFF) architecture is defined in section 6.2 of OAuth for Browser-Based Apps. In the BFF architecture, tokens are retrieved from the authorization server by using the backend and are securely stored in the backend. This process is achieved through the BFF proxy, which is a confidential client backend application that initiates the authorization code flow and stores access tokens and refresh tokens. When the flow successfully finishes, the BFF proxy issues a traditional browser session cookie to the single-page app, and the tokens never reach the app. All the requests to the resource server (the API that the app needs to call) go through the BFF proxy. The proxy uses the access token to make the request to the API and sends the response back to the app.

BFF has several advantages. The communication between the app and the proxy is hardened by various browser-level protection mechanisms (for example http-only, secure cookies) so that JavaScript can't read it. Additionally, the access token and refresh token aren't accessible from the JavaScript running in the front end because they never leave the backend proxy, thereby mitigating the JavaScript XSS attacks.

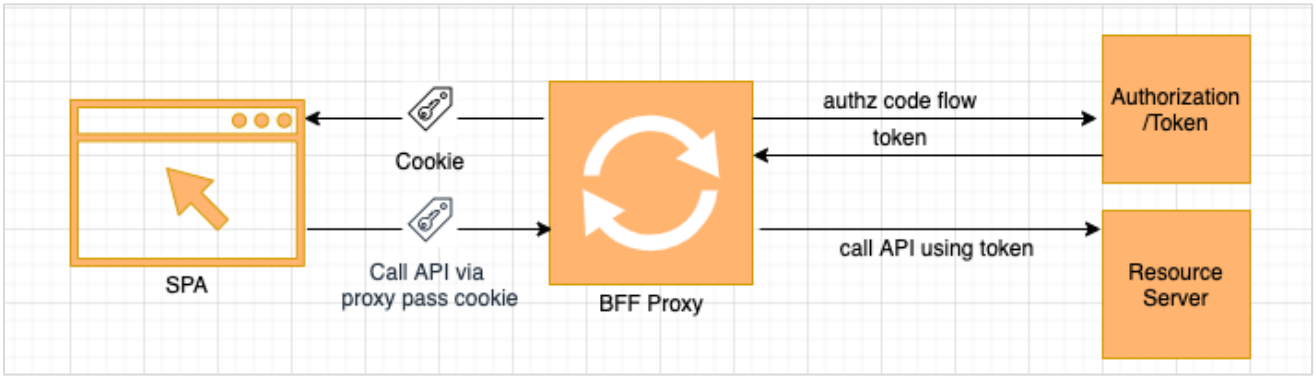

Figure 4: Authorization Flow with BFF Proxy

Key Best Practices

- Use the authorization code flow with PKCE.
- Keep access tokens short-lived and, if using refresh tokens, rotate them and invalidate them on user authentication state changes such as logout and session expiry.
- Don't store any sensitive information such as tokens in the browser. Use an architecture such as BFF so that tokens are never stored in the front end.

Best Practices for Native Mobile Apps

Native mobile apps are applications that are installed on a user's mobile device and run in the context of the device's OS. Unlike web-based applications, these apps don't run in a browser.

Public Clients

Like single-page apps, native mobile apps are also considered public clients because these apps are distributed using an app store. They can't have individual client secrets for each user who installs the app. Therefore, client authentication isn't recommended for native mobile apps.

Authorization Code with PKCE Flow

In the PKCE flow, an authorization code is returned to the app using the redirect URI. In native mobile apps, the redirect URI is defined at the app level. As a result, a malicious app can use the same redirect URI of a legitimate app and intercept the code returned from the authorization endpoint. Using the PKCE flow prevents this risk because even if the malicious app has the code, when it exchanges the code for a token, it must also pass a secret (code verifier) that it doesn't have. OCI IAM rejects the request and doesn't issue tokens.

Like single-page apps, the authorization code with PKCE flow is recommended for native mobile apps to prevent authorization code interception attacks.

External User Agents

As part of running the PKCE flow, a native mobile app sends a request to the authorization server by using a user agent such as a system browser or OS-specific user agents, or an embedded browser. Using embedded user agents, such as an embedded browser, or their implementation, such as WebView, poses a security risk. WebView runs in the context of the app, and the app can potentially track user keystrokes and get the user's credentials. Also, using WebView authentication is tied to the app and other apps can't participate in single sign-on.

The best practice is to use the following external agents for native mobile apps:

MOBILE OS	EXTERNAL AGENT
Android	Android Custom Tabs
iOS	iOS ASWebAuthenticationSession
Android and iOS	System browser

Using external agents such as a system browser is more secure and provides the following advantages:

- Users can see where they're entering the credentials and ensure it's the trusted IdP. This ensures user privacy.
- When users authenticate by using a system browser, a session cookie is created in the browser. Other mobile apps using authentication with the same IdP don't have to authenticate again because cookies are shared, thus achieving single sign-on.

Best Practices for Token Validation

Use these best practices for ID tokens and access tokens.

ID Tokens

ID tokens are meant for the client only. A client application must validate an ID token received from the IdP before using it in the application. OpenID Connect Core section 3.1.3.7 defines server rules for validating tokens.

- Verify that the issuer in the ID token matches the issuer of the IdP, and that the issuer URL is using HTTPS, isn't empty, and isn't using any query parameters. The client can get the issuer information of the IdP from the discovery endpoint.
- Verify that the token is intended for the client by checking the `aud` claim and verifying that it matches the `client_id`.
- Verify the token's signature, its expiry, and when it was issued. Obtain the identity domain's public key by calling the `jwks_uri` endpoint. Use the discovery endpoint to get the `jwks_uri` endpoint.
- Verify that the `nonce` value matches when the ID token was received as part of authorization code flow.

Access Token

Client applications use access tokens to call resource server APIs. Access tokens are meant for resource servers only, and the resource server must validate the access token before using it in the application.

- Verify that the issuer in the access token matches the issuer of the IdP, and that the issuer URL is using HTTPS, isn't empty, and isn't using any query parameters. The resource server can get the issuer information of the IdP from the discovery endpoint.
- Verify the token's signature, its expiry, and when it was issued. Obtain the identity domain's public key by calling the `jwks_uri` endpoint. Use the discovery endpoint to get the `jwks_uri` endpoint.
- Verify that the access token is intended for the resource server by checking the `aud` claim.
- Verify that the scope claim in the access token matches the resource or API that the client is accessing.

You can also validate the ID token and access token by calling the OCI IAM `introspect` endpoint, but this incurs an additional call compared to validating the token locally within the client.

General Best Practices

Follow these general best practices when using OAuth/OpenID Connect in OCI IAM.

Client Secrets

Store and retrieve client secrets securely. Avoid storing client secrets in flat files or configuration. Consider using the OCI Vault key management service. Rotate client secrets periodically based on your compliance requirement.

Application Roles and Scope

When configuring applications in OCI IAM, follow the principle of least privilege when assigning application roles or scope. For example, if the client is an application that performs only user operations, consider adding only the User Admin application role. For more information about application roles and API access mapping, see AppRole Permissions.

Client applications requesting scope should request the minimal scope needed. However, in the user can deny consent if additional scopes are requested.

Access Token and Refresh Token Expiry

Access tokens are bearer tokens. Anyone who has access to one can call the APIs that it's scoped to access. Also, because the resource server validates the access token, the access token can't be revoked from OCI IAM. Best practices are to use short-lived access tokens, use long-lived refresh tokens to get new access tokens, and set up timely revocation of refresh tokens. Even if you configure a longer-lived access token, OCI IAM has certain mechanisms to restrict token expiry based on several factors. For information, see Token Expiry Table.

Client Authentication

Typically, a shared secret (`client_secret`) is used for client authentication. However, for sensitive applications, consider using JWT client assertion (`private_key_jwt`) or TLS client authentication for client authentication.

Transport Layer Security (TLS)

All OCI IAM endpoints use HTTPS. Ensure that the redirect URLs configured in OCI IAM for clients also use HTTPS. By default, OCI IAM doesn't allow specifying a non-HTTPS URL. Also, ensure that communication to the resource server is secured through HTTPS.

Discovery URL

Use the OCI IAM discovery endpoint URL to obtain various endpoint URLs instead of individually storing endpoints in the configuration file. By default, OCI IAM doesn't enable public access to the discovery URL; however, it can be configured to be public.

User Groups

When you're using an OpenID Connect flow, often there's a need for the application to get user groups. In OCI IAM, you can request groups as part of the ID token or get an access token that's eligible to fetch user groups. When you request `scope=get_groups` in a request to the authorization endpoint, OCI IAM returns the user groups as part of the ID token, which increases the size of the ID token. An alternative is to use `scope=groups`, which allows the returned access token to get user groups from the `userinfo` endpoints. So, `scope=get_groups` saves an API call but increases the ID token size.

Logouts

When a client or application initiates a logout request to the IdP logout endpoint, OCI IAM logs out the user from OCI IAM and removes associated cookies. However, the client session, the access token, and the refresh token obtained by the client might still be active. Access tokens are bearer tokens and can therefore be used to access the resource even though the user is logged out of OCI IAM. To accomplish a complete logout, the best practice is to delete the access token and revoke the refresh token.

Example Requests and Responses

- Client credentials
- Authorization code flow
- Revoke refresh token
- Validate token
- OpenID Connect with code flow
- Device code flow
- User/client assertion
- TLS client authentication

References

This paper summarizes some of the topics from various OAuth/OpenID Connect-related specifications. For more information about OAuth/OpenID Connect security, see the following specifications:

- OAuth 2.0 Authorization Framework
- OpenID Connect Core specification
- OpenID Token Validations
- OAuth 2.0 for Browser-Based Apps
- JSON Web Token (JWT)
- OAuth 2.0 Security Best Current Practice
- Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants
- JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants