**API SECURITY AND AUTHENTICATION METHODS CHECK**

**VERIFACTS SERVICES PRIVATE LIMITED**

# Authentication Methods

# OAuth 2.0:

[OAuth2.0](https://oauth.net/2/) (hereinafter referred to as OAuth) is an authorization framework that allows a client to access resources on the behalf of its user.

In order to achieve this, OAuth heavily relies on tokens to communicate between the different entities, each entity having a different [role](https://datatracker.ietf.org/doc/html/rfc6749#section-1.1):

* Resource Owner: The entity who grants access to a resource, the owner, and in most cases is the user themselves
* Client: The application that is requesting access to a resource on behalf of the Resource Owner. These clients come in two [types](https://oauth.net/2/client-types/):
  + Public: clients that can’t protect a secret (e.g. frontend focused applications, such as SPAs, mobile applications, etc.)
  + Confidential: clients that are able to securely authenticate with the authorization server by keeping their registered secrets safe (e.g. backend services)
* Authorization Server: The server that holds authorization information and grants the access
* Resource Server: The application that serves the content accessed by the client

Since OAuth’s responsibility is to delegate access rights by the owner to the client, this is a very attractive target for attackers, and bad implementations lead to unauthorized access to the users’ resources and information.

In order to provide access to a client application, OAuth relies on several [authorization grant types](https://oauth.net/2/grant-types/) to generate an access token:

* [Authorization Code](https://oauth.net/2/grant-types/authorization-code/): used by both confidential and public clients to exchange an authorization code for an access token, but recommended only for confidential clients
* [Proof Key for Code Exchange (PKCE)](https://oauth.net/2/pkce/): PKCE builds on top of the Authorization Code grant, providing stronger security for it to be used by public clients, and improving the posture of confidential ones
* [Client Credentials](https://oauth.net/2/grant-types/client-credentials/): used for machine to machine communication, where the “user” here is the machine requesting access to its own resources from the Resource Server
* [Device Code](https://oauth.net/2/grant-types/device-code/): used for devices with limited input capabilities.
* [Refresh Token](https://oauth.net/2/grant-types/refresh-token/): tokens provided by the authorization server to allow clients to refresh users’ access tokens once they become invalid or expire. This grant type is used in conjunction with one other grant type.

Two flows will be deprecated in the release of [OAuth2.1](https://oauth.net/2.1/), and their usage is not recommended:

* [Implicit Flow\*](https://oauth.net/2/grant-types/implicit/): PKCE’s secure implementation renders this flow obsolete. Prior to PKCE, the implicit flow was used by client-side applications such as [single page applications](https://en.wikipedia.org/wiki/Single-page_application) since [CORS](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS) relaxed the [same-origin policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy) for sites to inter-communicate.
* [Resource Owner Password Credentials](https://oauth.net/2/grant-types/password/): used to exchange users’ credentials directly with the client, which then sends them to the authorization to exchange them for an access token.

## **How to Test:**

### Testing for Deprecated Grant Types:

Deprecated grant types were obsoleted for security and functionality reasons. Identifying if they’re being used allows us to quickly review if they’re susceptible to any of the threats pertaining to their usage. Some might be out of scope to the attacker, such as the way a client might be using the users’ credentials. This should be documented and raised to the internal engineering teams.

For public clients, it is generally possible to identify the grant type in the request to the /token endpoint. It is indicated in the token exchange with the parameter grant\_type.

The following example shows the Authorization Code grant with PKCE.

POST /oauth/token HTTP/1.1

Host: as.example.com

[...]

{

"client\_id":"example-client",

"code\_verifier":"example",

"grant\_type":"authorization\_code",

"code":"example",

"redirect\_uri":"http://client.example.com"

}

The values for the grant\_type parameter and the grant type they indicate are:

* password: Indicates the ROPC grant.
* client\_credentials: Indicates the Client Credential grant.
* authorization\_code: Indicates the Authorization Code grant.

The Implicit Flow type is not indicated by the grant\_type parameter since the token is presented in the response to the /authorization endpoint request, and instead can be identified through the response\_type. Below is an example.

GET /authorize

?client\_id=<some\_client\_id>

&response\_type=token

&redirect\_uri=https%3A%2F%2Fclient.example.com%2F

&scope=openid%20profile%20email

&state=<random\_state>

The following URL parameters indicate the OAuth flow being used:

* response\_type=token: Indicates Implicit Flow, as the client is directly requesting from the authorization server to return a token.
* response\_type=code: Indicates Authorization Code flow, as the client is requesting from the authorization server to return a code, that will be exchanged afterwards with a token.
* code\_challenge=sha256(xyz): Indicates the PKCE extension, as no other flow uses this parameter.

The following is an example authorization request for Authorization Code flow with PKCE:

GET /authorize

?redirect\_uri=https%3A%2F%2Fclient.example.com%2F

&client\_id=<some\_client\_id>

&scope=openid%20profile%20email

&response\_type=code

&response\_mode=query

&state=<random\_state>

&nonce=<random\_nonce>

&code\_challenge=<random\_code\_challenge>

&code\_challenge\_method=S256 HTTP/1.1

Host: as.example.com

[...]

### Public Clients:

The Authorization Code grant with PKCE extension is recommended for public clients. An authorization request for Authorization Code flow with PKCE should contain response\_type=code and code\_challenge=sha256(xyz).

The token exchange should contain the grant type authorization\_code and a code\_verifier.

Improper grant types for public clients are:

* Authorization Code grant without the PKCE extension
* Client Credentials
* Implicit Flow
* ROPC

### Confidential Clients:

The Authorization Code grant is recommended for confidential clients. The PKCE extension may be used as well.

Improper grant types for confidential clients are:

* Client Credentials (Except for machine-to-machine – see below)
* Implicit Flow
* ROPC

### Machine-to-Machine:

In situations where no user interaction occurs and the clients are only confidential clients, the Client Credentials grant may be used.

If you know the client\_id and client\_secret, it is possible to obtain a token by passing the client\_credentials grant type.

$ curl --request POST \

--url https://as.example.com/oauth/token \

--header 'content-type: application/json' \

--data '{"client\_id":"<some\_client\_id>","client\_secret":"<some\_client\_secret>","grant\_type":"client\_credentials"}' --proxy http://localhost:8080/ -k

### Credential Leakage:

Depending on the flow, OAuth transports several types of credentials in as URL parameters.

The following tokens can be considered to be leaked credentials:

* access token
* refresh token
* authorization code

PKCE code challenge / code verifier

Due to how OAuth works, the authorization code as well as the code\_challenge, and code\_verifier may be part of the URL. The implicit flow transports the authorization token as part of the URL if the response\_mode is not set to [form\_post](https://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html). This may lead to leakage of the requested token or code in the referrer header, in log files, and proxies due to these parameters being passed either in the query or the fragment.

The risk that’s carried by the implicit flow leaking the tokens is far higher than leaking the code or any other code\_\* parameters, as they are bound to specific clients and are harder to abuse in case of leakage.

In order to test this scenario, make use of an HTTP intercepting proxy such as ZAP and intercept the OAuth traffic.

* Step through the authorization process and identify any credentials present in the URL.
* If any external resources are included in a page involved with the OAuth flow, analyze the request made to them. Credentials could be leaked in the referrer header.

After stepping through the OAuth flow and using the application, a few requests are captured in the request history of an HTTP intercepting proxy. Search for the HTTP referrer header (e.g. Referer: https://idp.example.com/) containing the authorization server and client URL in the request history.

Reviewing the HTML meta tags (although this tag is [not supported](https://caniuse.com/mdn-html_elements_meta_name_referrer) on all browsers), or the [Referrer-Policy](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy) could help assess if any credential leakage is happening through the referrer header.

# API Keys:

API keys are supplied by client users and applications calling REST APIs to track and control how the APIs are used (for example, to meter access and prevent abuse or malicious attack). The Authenticate API Key filter enables you to securely authenticate an API key with the API Gateway. API keys include a key ID that identifies the client responsible for the API service request. This key ID is not a secret, and must be included in each request. API keys can also include a confidential secret key used for authentication, which should only be known to the client and to the API service. You can use the Authenticate API Key filter to specify where to find the API key ID and secret key in the request message, and to specify timestamp and expiry options.

An example use case for this filter would be a client accessing a REST API service to invoke specific methods (for example, startVM() or stopVM()). To invoke these methods, you are required to provide your API key ID and secret key to the API Gateway. You can keep the secret key private by sending the request over HTTPS. Alternatively, you can use the secret key to generate an HMAC digital signature. This means that the secret key is not sent in the request, but is inferred instead, because the message must have been signed using the required secret key. When the API service receives the request, it uses the API key ID to look up the corresponding secret key, and uses it to validate the signature and confirm the request sender.

The API Gateway supports the following API key types:

* Simple API keys including a key ID only. The API key ID is included in all requests to authenticate the client.
* Amazon Web Services style API keys including a key ID and a secret key, which are used together to securely authenticate the client. The API key ID is included in all requests to identify the client. The secret key is known only to the client and the API Gateway.

## **How to Pass API Keys in Headers**

If you determine that the simplicity of API keys makes sense for your use case, there are several places where API keys can be passed in your [API design](https://stoplight.io/api-design-guide/basics/).

The most popular API key location for modern APIs is in headers. However, that’s not enough information: where in the headers should you include the API key? There are several methods that we’ll cover next.

Before that, an important note: as with all API requests, use HTTPS (TLS, the successor to SSL) to ensure that data is encrypted in transit.

### x-api-key

The most popular choice for including API keys in headers, perhaps due to its usage by AWS API Gateway, `x-api-key` is a custom header convention for passing your API key. For more on API gateway authentication, [see this post about API gateways.](https://blog.stoplight.io/api-proxy-vs-api-gateway-c008c942a02d)

**GET** / HTTP/1.1

Host: example.com

X-API-**KEY**: abcdef12345

### Basic Authentication

How long should an API key be? It depends. Earlier, we suggested Basic Auth as an alternative to API keys. Basic Auth and API keys can also be used together. You can pass the API key via Basic Auth as either the username or password. Most implementations pair the API key with a blank value for the unused field (username or password).

**GET** / HTTP/1.1

Host: example.com

Authorization: Basic bWFnZ2llOnN1bW1lcnM=

You will need to base64-encode the `username:password` content, but most request libraries do this for you.

### Bearer Authentication

Some APIs use the `Authorization` header to handle the API key, usually with the Bearer keyword. This method is also used for other tokens, such as those generated by OAuth.

The client must send this token in the `Authorization` header when making requests to protected resources:

Authorization: Bearer abcdef12345

What about non-header locations for API keys? You can find them in query strings or even the data body.

### Other API Key Locations

Though the header has become the preferred location for API keys, there are non-header methods still used by many APIs. As a developer using APIs, you may spot these methods in the wild. As an API designer, you’ll probably want to stick to the headers, as we’ll explain.

### Query String

A popular method for early APIs, passing an API key through a query string in a URL is certainly easy. However, this method can risk API key exposure since, despite encryption, the parameters can be stored in web server logs.

curl -X **GET** "https://example.com/endpoint/?api\_key=abcdef12345"

If you use the query string method, you’ll want to make sure that there’s a low risk of the API key being shared.

### Request Body Parameter

Another method we’ve seen, especially in older APIs, is to pass an API key in the POST body as JSON:

curl -X POST

`https://example.com/endpoint/’

-H ‘content-type: application/json’

-d ‘ {

“api\_key”: abcdef12345”

}’

The most significant drawback to this method is that authentication is mixed in with other data. It also encourages poor REST practices, as simple reads from the API would need to be sent a POST request instead of GET.

### JavaScript API

Finally, you may see API keys used with front-end JavaScript APIs, which provide in-browser access to API functionality. In these cases, the API key is passed one of two ways. Either the key is passed with the call to the script or in the JavaScript itself.

For example, Google Maps passes the key in the query string to the JavaScript:

<**script** async defer src="https://maps.googleapis.com/maps/api/js?key=YOUR\_API\_KEY&callback=initMap" type="text/javascript"></**script**>

Keen Dataviz, on the other hand, passes the API in a constructor:

**const** client = **new** KeenAnalysis({

projectId: 'YOUR\_PROJECT\_ID',

readKey: 'YOUR\_READ\_KEY'

});

In both cases, the companies take additional steps to secure the API calls. This is important because the API keys are essentially public, as they are easily discoverable if you view the source. Google Maps allows developers to restrict its usage on certain websites. Keen has separate read and write API keys.

### API Key Authentication using OpenAPI

In addition to human-readable API documentation, an OpenAPI definition is a must when designing APIs. You can describe your entire API in a machine-readable file (YAML or JSON). The format is meant to cover the many ways developers create RESTful APIs and support security schemes, including API keys, so it is flexible enough for any of the methods we’ve discussed.

For example, here is the security section of Stripe’s OpenAPI document, showing the two header approaches supported for its API keys:

securitySchemes:

basicAuth:

description: 'Basic HTTP authentication. Allowed headers-- Authorization: Basic

<api\_key> | Authorization: Basic <base64 hash **of** `api\_key:`>' scheme: basic type: http bearerAuth: bearerFormat: auth-scheme description: 'Bearer HTTP authentication. Allowed headers-- Authorization: Bearer <api\_key>' scheme: bearer type: http

A machine-readable API specification allows you to test the implementation against the specification throughout your API development lifecycle without extensive effort.

This quick tour should be just enough to get you started with API key authentication but see our longer series on [planning](https://blog.stoplight.io/plan-your-api-auth-strategy-with-careful-questions-auth-part-2) and [designing](https://blog.stoplight.io/design-first-auth-choose-a-strategy-that-works-auth-part-3) your API authentication strategy for a more comprehensive view.

[Stoplight Platform](https://docs.stoplight.io/docs/platform/52ab0a117eadd-welcome-to-the-stoplight-docs) makes it easy to design your API visually—including security definitions around any authentication—with OpenAPI. We hope you enjoyed these API authentication and authorization tips!

# JWT (JSON Web Tokens):

JSON Web Tokens (JWTs) are cryptographically signed JSON tokens, intended to share claims between systems. They are frequently used as authentication or session tokens, particularly on REST APIs.

JWTs are a common source of vulnerabilities, both in how they are in implemented in applications, and in the underlying libraries. As they are used for authentication, a vulnerability can easily result in a complete compromise of the application.

JWTs are made up of three components:

* The header
* The payload (or body)
* The signature

Each component is base64 encoded, and they are separated by periods (.).

### Header:

The header defines the type of token (typically JWT), and the algorithm used for the signature. An example decoded header is shown below:

{

"alg": "HS256",

"typ": "JWT"

}

There are three main types of algorithms that are used to calculate the signatures:

| Algorithm | Description |
| --- | --- |
| HSxxx | HMAC using a secret key and SHA-xxx. |
| RSxxx and PSxxx | Public key signature using RSA. |
| ESxxx | Public key signature using ECDSA. |

There are also a wide range of [other algorithms](https://www.iana.org/assignments/jose/jose.xhtml#web-signature-encryption-algorithms) which may be used for encrypted tokens (JWEs), although these are less common.

### Payload:

The payload of the JWT contains the actual data. An example payload is shown below:

{

"username": "administrator",

"is\_admin": true,

"iat": 1516239022,

"exp": 1516242622

}

The payload is it not usually encrypted, so review it to determine whether there is any sensitive of potentially inappropriate data included within it.

This JWT includes the username and administrative status of the user, as well as two standard claims (iat and exp). These claims are defined in [RFC 5719](https://tools.ietf.org/html/rfc7519#section-4.1), a brief summary of them is given in the table below:

| **Claim** | **Full Name** | **Description** |
| --- | --- | --- |
| iss | Issuer | The identity of the party who issued the token. |
| iat | Issued At | The Unix timestamp of when the token was issued. |
| nbf | Not Before | The Unix timestamp of earliest date that the token can be used. |
| exp | Expires | The Unix timestamp of when the token expires. |

### Signature:

The signature is calculated using the algorithm defined in the JWT header, and then base64 encoded and appended to the token. Modifying any part of the JWT should cause the signature to be invalid, and the token to be rejected by the server.

### Review Usage:

As well as being cryptographically secure itself, the JWT also needs to be stored and sent in a secure manner. This should include checks that:

* It is always [sent over encrypted (HTTPS) connections](https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/09-Testing_for_Weak_Cryptography/03-Testing_for_Sensitive_Information_Sent_via_Unencrypted_Channels).
* If it is stored in a cookie, then it should be [marked with appropriate attributes](https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/06-Session_Management_Testing/02-Testing_for_Cookies_Attributes).

The validity of the JWT should also be reviewed, based on the iat, nbf and exp claims, to determine that:

* The JWT has a reasonable lifespan for the application.
* Expired tokens are rejected by the application.

### Signature Verification:

One of the most serious vulnerabilities encountered with JWTs is when the application fails to validate that the signature is correct. This usually occurs when a developer uses a function such as the NodeJS jwt.decode() function, which simply decodes the body of the JWT, rather than jwt.verify(), which verifies the signature before decoding the JWT.

This can be easily tested for by modifying the body of the JWT without changing anything in the header or signature, submitting it in a request to see if the application accepts it.

# OpenID Connect:

OpenId is an HTTP-based protocol that uses identity providers to validate that a user is who they say they are. It is a very simple protocol which allows a service provider initiated way for single sign-on (SSO). This allows the user to re-use a single identity given to a trusted OpenId identity provider and be the same user in multiple websites, without the need to provide any website with the password, except for the OpenId identity provider.

Due to its simplicity and that it provides protection of passwords, OpenId has been well adopted. Some of the well-known identity providers for OpenId are Stack Exchange, Google, Facebook and Yahoo!

For non-enterprise environments, OpenId is considered a secure and often better choice, as long as the identity provider is of trust.

We present here how to test the OpenID Connect protocol (authorization code flow) with commande line tools, like curl. We use in this example a public OIDC provider based on LL::NG: [https://oidctest.wsweet.org](https://oidctest.wsweet.org/)

## **Authentication**

The first step is to obtain a valid SSO session on the portal. The standard solution is to use a web browser and log into the portal, then get the value of the SSO cookie.

In our case, to be able to use only command lines, we will use portal REST API (which requires to adapt the requireToken configuration to get cookie value in JSON response (see [REST services](https://lemonldap-ng.org/documentation/2.0/restservices.html)). This should not be what you want on a production service.

Example of REST service usage, with credentials dwho/dwho:

curl -X POST -d user=dwho -d password=dwho -H 'Accept: application/json' 'https://oidctest.wsweet.org/oauth2/'

The session id is displayed in JSON response:

{

"error" : "0",

"id" : "0640f95827111f00ba7ad5863ba819fe46cfbcecdb18ce525836369fb4c8350b",

"result" : 1

}

## **Authorization code**

In the first step of authorization code flow, we request a temporary code, on the authorize end point.

**Required parameters:**

* SSO session id (will be passed in lemonldap cookie, adapt the name if needed)
* Client ID: given by your OIDC provider, we use here private
* Scope: depends on which information you want, we will use here openid profile email
* Redirect URI: shoud match the value registered in your OIDC provider, we will use here http://localhost

The OIDC provide will return the code in the location header, so we just output this reponse header:

curl -s -D - -o /dev/null -b lemonldap=0640f95827111f00ba7ad5863ba819fe46cfbcecdb18ce525836369fb4c8350b 'https://oidctest.wsweet.org/oauth2/authorize?response\_type=code&client\_id=private&scope=openid+profile+email&redirect\_uri=http://localhost' | grep -i '^location'

The value of the location header is:

location: http://localhost?code=294b0facd91a0fa92762edc48d18369e99c330ba2b8fb05ab2c45999fcef6e17&session\_state=BpB8KRMBEDUs%2B7lAjsz4DRk3E0RJImxgUbMsCFFAUa8%3D.N3dVOFg3a2RpNXVJK3ltSldrYXZjUjhtU0tvd29sWkpuWWJJbll5ZGs5NzhZMnh5bmQwd0IxRmJVWUxJSTlkWDBnSWZ2SWFVZmU0UnRaMkVJVjNUY3c9PQ

So we get the code value: 294b0facd91a0fa92762edc48d18369e99c330ba2b8fb05ab2c45999fcef6e17  
This code has a short lifetime, we will use it to get access token and ID token in the next step

## **Tokens**

**In this step, we exchange the authorization code against tokens:**

* Access token
* ID token
* Refresh token (optional)

**Required parameters:**

* Authorization code: see previous step
* Grant type: we use here authorization\_code
* Redirect URI: same value as the one used in the previous step
* Client ID and Client Secret: given by your OIDC provider, we use here private/tardis

curl -s -X POST -d grant\_type=authorization\_code -d 'redirect\_uri=http://localhost' -d code=294b0facd91a0fa92762edc48d18369e99c330ba2b8fb05ab2c45999fcef6e17 -u 'private:tardis' 'https://oidctest.wsweet.org/oauth2/token' | json\_pp

The JSON response looks like this:

{

"access\_token" : "a88b8dde538719e55c3cb8fbd14d06ed77853c685a62abf6ecb88d86228a9c64",

"expires\_in" : 3600,

"id\_token" : "eyJhbGciOiJSUzI1NiIsImtpZCI6Im9pZGN0ZXN0IiwidHlwIjoiSldUIn0.eyJhdXRoX3RpbWUiOjE2MTQxNjAwMDYsImlhdCI6MTYxNDE2MzIxOCwiaXNzIjoiaHR0cHM6Ly9vaWRjdGVzdC53c3dlZXQub3JnLyIsImF0X2hhc2giOiJIVGswOVNjSjRObEFua3k5SGFFX2VRIiwiYWNyIjoibG9hLTIiLCJleHAiOjE2MTQxNjY4MTgsInN1YiI6ImR3aG8iLCJhenAiOiJwcml2YXRlIiwiYXVkIjpbInByaXZhdGUiXX0.N3TNufjKLzKM3qiIitA7JHUei4L572XjF6AcVl7UAFB6efdGUCiAL7amlUl0FgjZfzW9bzvulBVDidoYSicIaysIdI4KkjmjpVN0Z3gOSu0ecuk5p8fD1KbX6-tmA3txeR18nzfhdckq-S-6Lx7wrWpPNyrzGx-FImbOaUPN2yeVhKPXhdyHJbzI0RqJETxnBkyW-CLEzAJyq3rCUVX-D8kHADvg6a42QQyPdxvBuGrdBfyDDDb\_Py13H1qhn40NnuFknR1wSahsY6U97uUooyk-0\_U4J3XJAHySjCtivtSeP0fM\_5eblMuh6WdVjrfnUF0xnCTbCa2gYRlTS38BkqcsWY26PXoRAOo31a1cmB5sMSZyPtRF9UZcmGiNBIymMMdFgVAJONb6uliiTS5j9-nkmHOqVC-XJ6tuiU3ZSBQ8nCRyNW2LaCzpJ5c3ytP9yYQtyT8HmhN0VnXob3K1uJEA\_Xcu4sADjtrm-LbrGiwaVMkfu-C6YIrbuC9riOW6TneV2gAzAjXPOW\_UZeXrCrx66GHIJPsJIq29UfbTN5Pxo9SH2yKw6PSfxevkZhBIhEXCOMaIUHrlWz2jDBBzPIWeiSRbK\_MRtejQmdRUs8nqdq-McVwnFiUMDt1KZXxqScTtMDF\_Lo9oK2RaCijEJ7MSPEscr\_YOyp3KIq2FLVg",

"refresh\_token" : "19434440ed4da2803e8ba9d91cb2eabd5b8bd12af2609429bda03ed487e6ef57",

"token\_type" : "Bearer"

}

The access token will be used for the last step, to retrieve information about the user.

The ID Token is a JWT (JSON Web Token) and can be parsed easily, as this is the concatenation of 3 JSON strings encoded in base 64: base64(header).base64(payload).base64(signature).

Decoding the payload gives:

echo 'eyJhdXRoX3RpbWUiOjE2MTQxNjAwMDYsImlhdCI6MTYxNDE2MzIxOCwiaXNzIjoiaHR0cHM6Ly9vaWRjdGVzdC53c3dlZXQub3JnLyIsImF0X2hhc2giOiJIVGswOVNjSjRObEFua3k5SGFFX2VRIiwiYWNyIjoibG9hLTIiLCJleHAiOjE2MTQxNjY4MTgsInN1YiI6ImR3aG8iLCJhenAiOiJwcml2YXRlIiwiYXVkIjpbInByaXZhdGUiXX0=' | base64 -d | json\_pp

{

"acr" : "loa-2",

"at\_hash" : "HTk09ScJ4NlAnky9HaE\_eQ",

"aud" : [

"private"

],

"auth\_time" : 1614160006,

"azp" : "private",

"exp" : 1614166818,

"iat" : 1614163218,

"iss" : "https://oidctest.wsweet.org/",

"sub" : "dwho"

}

## **User info**

This step is optional and allows to fetch user information linked to scopes requested in the first step.

**Required parameters:**

* Access token, used as bearer authorization

curl -s -H 'Authorization: Bearer a88b8dde538719e55c3cb8fbd14d06ed77853c685a62abf6ecb88d86228a9c64' 'https://oidctest.wsweet.org/oauth2/userinfo' | json\_pp

JSON response:

{

"email" : "dwho@badwolf.org",

"name" : "Doctor Who",

"preferred\_username" : "dwho",

"sub" : "dwho"

}

## **Introspection**

You can test access token validity with the introspection endpoint.

**Required parameters:**

* Client ID and Client Secret, used as basic authorization
* Access token, sent as POST data

curl -s -u private:tardis -X POST -d 'token=a88b8dde538719e55c3cb8fbd14d06ed77853c685a62abf6ecb88d86228a9c64' 'https://oidctest.wsweet.org/oauth2/introspect' | json\_pp

JSON response:

{

"active" : **true**,

"client\_id" : "private",

"exp" : 1630684115,

"iss" : "https://oidctest.wsweet.org/",

"scope" : "openid profile email",

"sub" : "dwho"

}

## **Refresh an access token**

If the access token has expired, you can get a new one with the refresh token.

**Required parameters:**

* Grant type: we use here refresh\_token, sent as POST data
* Refresh token, sent as POST data
* Client ID and Client Secret, used as basic authorization

curl -s -X POST -d grant\_type=refresh\_token -d refresh\_token=19434440ed4da2803e8ba9d91cb2eabd5b8bd12af2609429bda03ed487e6ef57 -u 'private:tardis' 'https://oidctest.wsweet.org/oauth2/token' | json\_pp

JSON response:

{

"access\_token" : "78929118546b1a11a2e3b607f607d0ccb73d72bbd95c59d0b03ae69ffa17f41a",

"expires\_in" : 3600,

"id\_token" : "eyJhbGciOiJSUzI1NiIsImtpZCI6Im9pZGN0ZXN0IiwidHlwIjoiSldUIn0.eyJhdXRoX3RpbWUiOjE2MTQxNjAwMDYsImlhdCI6MTYxNDE2MzIxOCwiaXNzIjoiaHR0cHM6Ly9vaWRjdGVzdC53c3dlZXQub3JnLyIsImF0X2hhc2giOiJIVGswOVNjSjRObEFua3k5SGFFX2VRIiwiYWNyIjoibG9hLTIiLCJleHAiOjE2MTQxNjY4MTgsInN1YiI6ImR3aG8iLCJhenAiOiJwcml2YXRlIiwiYXVkIjpbInByaXZhdGUiXX0.N3TNufjKLzKM3qiIitA7JHUei4L572XjF6AcVl7UAFB6efdGUCiAL7amlUl0FgjZfzW9bzvulBVDidoYSicIaysIdI4KkjmjpVN0Z3gOSu0ecuk5p8fD1KbX6-tmA3txeR18nzfhdckq-S-6Lx7wrWpPNyrzGx-FImbOaUPN2yeVhKPXhdyHJbzI0RqJETxnBkyW-CLEzAJyq3rCUVX-D8kHADvg6a42QQyPdxvBuGrdBfyDDDb\_Py13H1qhn40NnuFknR1wSahsY6U97uUooyk-0\_U4J3XJAHySjCtivtSeP0fM\_5eblMuh6WdVjrfnUF0xnCTbCa2gYRlTS38BkqcsWY26PXoRAOo31a1cmB5sMSZyPtRF9UZcmGiNBIymMMdFgVAJONb6uliiTS5j9-nkmHOqVC-XJ6tuiU3ZSBQ8nCRyNW2LaCzpJ5c3ytP9yYQtyT8HmhN0VnXob3K1uJEA\_Xcu4sADjtrm-LbrGiwaVMkfu-C6YIrbuC9riOW6TneV2gAzAjXPOW\_UZeXrCrx66GHIJPsJIq29UfbTN5Pxo9SH2yKw6PSfxevkZhBIhEXCOMaIUHrlWz2jDBBzPIWeiSRbK\_MRtejQmdRUs8nqdq-McVwnFiUMDt1KZXxqScTtMDF\_Lo9oK2RaCijEJ7MSPEscr\_YOyp3KIq2FLVg",

"token\_type" : "Bearer"

}

## **Logout**

To kill SSO session, call the OIDC logout endpoint. By default a confirmation is requested, but you can bypass it by adding confirm=1 to URL.

**Required parameters:**

* SSO session id (will be passed in lemonldap cookie)

curl -s -D - -o /dev/null -b lemonldap=0640f95827111f00ba7ad5863ba819fe46cfbcecdb18ce525836369fb4c8350b 'https://oidctest.wsweet.org/oauth2/logout?confirm=1'

The session is deleted on server side and the cookie is destroyed in the browser. You can use the introspection endpoint to verify that the access token is no longer valid.