Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: November 10, 2010

E. Hammer-Lahav, Ed.
Yahoo!
D. Recordon
Facebook
D. Hardt

May 9, 2010

The OAuth 2.0 Protocol
draft-ietf-oauth-v2-04

Abstract

This specification describes the OAuth 2.0 protocol. OAuth provides a method for making authenticated HTTP requests using a token - an identifier used to denote an access grant with specific scope, duration, and other attributes. Tokens are issued to third-party clients by an authorization server with the approval of the resource owner. OAuth defines multiple flows for obtaining a token to support a wide range of client types and user experience.

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 http://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 November 10, 2010.

Copyright Notice

Table of Contents

1. Authors

This specification was authored with the participation and based on the work of Allen Tom (Yahoo!), Brian Eaton (Google), Brent Goldman (Facebook), Luke Shepard (Facebook), Raffi Krikorian (Twitter), and Yaron Goland (Microsoft).

2. Introduction

With the increasing use of distributed web services and cloud computing, third-party applications require access to server-hosted resources. These resources are usually protected and require authentication using the resource owner's credentials (typically a username and password). In the traditional client-server authentication model, a client accessing a protected resource on a server presents the resource owner's credentials in order to authenticate and gain access.

Resource owners should not be required to share their credentials when granting third-party applications access to their protected resources. They should also have the ability to restrict access to a limited subset of the resources they control, to limit access duration, or to limit access to the HTTP methods supported by these resources.

OAuth provides a method for making authenticated HTTP requests using a token - an identifier used to denote an access grant with specific scope, duration, and other attributes. Tokens are issued to third-

party clients by an authorization server with the approval of the resource owner. Instead of sharing their credentials with the client, resource owners grant access by authenticating directly with the authorization server which in turn issues a token to the client. The client uses the token (and optional secret) to authenticate with the resource server and gain access.

For example, a web user (resource owner) can grant a printing service (client) access to her protected photos stored at a photo sharing service (resource server), without sharing her username and password with the printing service. Instead, she authenticates directly with the photo sharing service (authorization server) which issues the printing service delegation-specific credentials (token).

This specification defines the use of OAuth over HTTP [RFC2616] (or HTTP over TLS 1.0 as defined by [RFC2818]. Other specifications may extend it for use with other transport protocols.

2.1. Terminology

resource server
: An HTTP [RFC2616] server capable of accepting authenticated resource requests using the OAuth protocol.

protected resource
: An access-restricted resource which can be obtained from a resource server using an OAuth-authenticated request.

client
: An HTTP client capable of making authenticated requests for protected resources using the OAuth protocol.

resource owner
: An entity capable of granting access to a protected resource.

end-user
: A human resource owner.

access token
: A unique identifier used by the client to make authenticated

requests on behalf of the resource owner. Access tokens may have a matching secret.

bearer token An access token without a matching secret, used to obtain access to a protected resource by simply presenting the access token as-is to the resource server.

authorization server
An HTTP server capable of issuing tokens after successfully authenticating the resource owner and obtaining authorization. The authorization server may be the same server as the resource server, or a separate entity.

authorization endpoint
The authorization server's HTTP endpoint capable of authenticating the resource owner and obtaining authorization.

token endpoint
The authorization server's HTTP endpoint capable of issuing tokens and refreshing expired tokens.

client identifier
An unique identifier issued to the client to identify itself to the authorization server. Client identifiers may have a matching secret.

refresh token
A unique identifier used by the client to replace an expired access token with a new access token without having to involve the resource owner. A refresh token is used when the access token is valid for a shorter time period than the duration of the access grant approved by the resource owner.

2.2. Overview

Clients interact with a protected resource, first by requesting access (which is granted in the form of an access token) from the authorization server, and then by authenticating with the resource server by presenting the access token. Figure 1 demonstrates the flow between the client and authorization server (A, B), and the flow between the client and resource server (C, D), when the client is

acting autonomously (the client is also the resource owner).

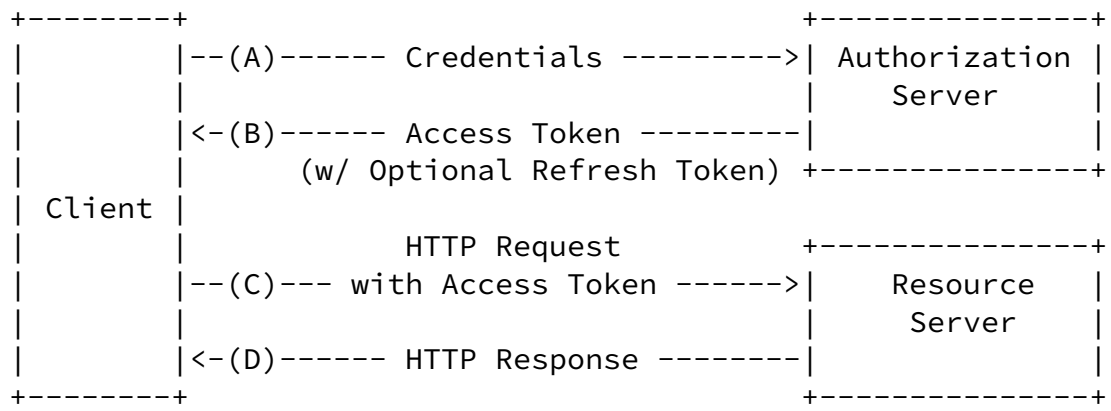
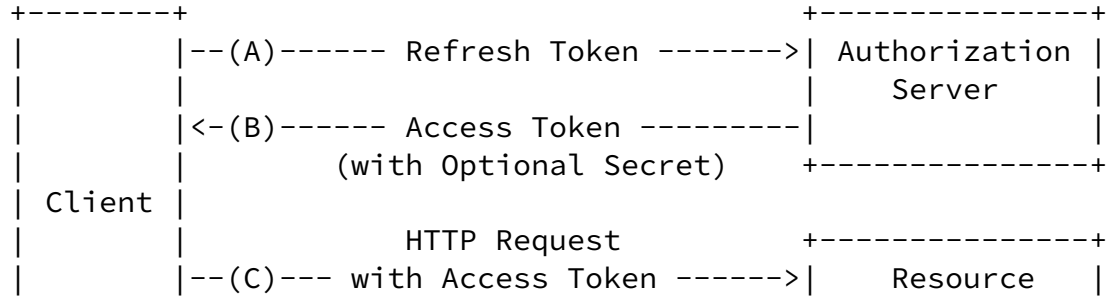


Figure 1

Access token strings can use any internal structure agreed upon between the authorization server and the resource server, but their structure is opaque to the client. Since the access token provides the client access to the protected resource for the life of the access token (or until revoked), the authorization server should issue access tokens which expire within an appropriate time, usually much shorter than the duration of the access grant.

When an access token expires, the client can request a new access token from the authorization server by presenting its credentials again (Figure 1), or by using the refresh token (if issued with the access token) as shown in Figure 2. Once an expired access token has been replaced with a new access token (A, B), the client uses the new access token as before (C, D).



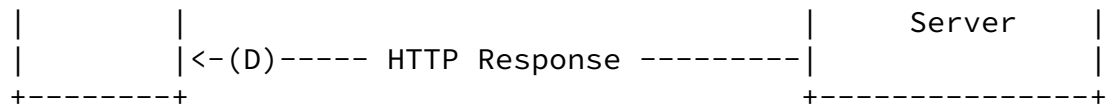


Figure 2

This specification defines a number of authorization flows to support different client types and scenarios. These authorization flows can be separated into three groups: user delegation flows, direct credentials flows, and autonomous flows.

Additional authorization flows may be defined by other specifications to cover different scenarios and client types.

User delegation flows are used to grant client access to protected resources by the end-user without sharing the end-user credentials (e.g. a username and password) with the client. Instead, the end-user authenticates directly with the authorization server, and grants client access to its protected resources. The user delegation flows defined by this specifications are:

- User-Agent Flow - This flow is designed for clients running inside a user-agent (typically a web browser). This flow is described in Section 3.5.

- Web Server Flow - This flow is optimized for clients that are part of a web server application, accessible via HTTP requests. This flow is described in Section 3.6.

- Device Flow - This flow is suitable for clients executing on limited devices, but where the end-user has separate access to a user-agent on another computer or device. This flow is described in Section 3.7.

Direct credentials flows enable clients to obtain an access token with a single request using the client credentials or end-user credentials without seeking additional resource owner authorization. The direct credentials flows defined by this specification are:

- Username and Password Flow - This flow is used in cases where the

end-user trusts the client to handle its credentials but it is still undesirable for the client to store the end-user's username and password. This flow is only suitable when there is a high degree of trust between the end-user and the client. This flow is described in Section 3.8.

- Client Credentials Flow - The client uses its credentials to obtain an access token. This flow is described in Section 3.9.

Autonomous flows enable clients to use utilize existing trust relationships or different authorization constructs to obtain an access token. They provide a bridge between OAuth and other trust frameworks. The autonomous authorization flow defined by this specifications is:

- Assertion Flow - The client presents an assertion such as a SAML [OASIS.saml-core-2.0-os] assertion to the authorization server in exchange for an access token. This flow is described in Section 3.10.

2.3. Example

[[Todo]]

2.4. Notational Conventions

The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'MAY', and 'OPTIONAL' in this document are to be interpreted as described in [RFC2119].

This document uses the Augmented Backus-Naur Form (ABNF) notation of [I-D.ietf-httpbis-p1-messaging]. Additionally, the realm and auth-param rules are included from [RFC2617], and the URI-Reference rule from [RFC3986].

2.5. Conformance

An implementation is not compliant if it fails to satisfy one or more of the MUST or REQUIRED level requirements for the flows it implements. An implementation that satisfies all the MUST or REQUIRED level and all the SHOULD level requirements for its flows is said to be "unconditionally compliant"; one that satisfies all the MUST level requirements but not all the SHOULD level requirements for its flows is said to be "conditionally compliant."

3. Obtaining an Access Token

The client obtains an access token by using one of the authorization flows supported by the authorization server. The authorization flows all use the same authorization and token endpoints, each with a different set of request parameters and values.

Access tokens have a scope, duration, and other access attributes granted by the resource owner. These attributes MUST be enforced by the resource server when receiving a protected resource request, and by the authorization server when receiving a token refresh request.

In many cases it is desirable to issue access tokens with a shorter lifetime than the duration of the authorization grant. However, it may be undesirable to require the resource owner to authorize the request again. Instead, the authorization server issues a refresh token in addition to the access token. When the access token expires, the client can request a new access token without involving the resource owner as long as the authorization grant is still valid. The token refresh method is described in Section 4.

3.1. Authorization Endpoint

Clients direct the resource owner to the authorization endpoint to approve their access request. Before granting access, the resource owner first authenticates with the authorization server. The way in which the authorization server authenticates the end-user (e.g. username and password login, OpenID, session cookies) and in which the authorization server obtains the end-user's authorization, including whether it uses a secure channel such as TLS/SSL, is beyond the scope of this specification. However, the authorization server MUST first verify the identity of the end-user.

The URI of the authorization endpoint can be found in the service documentation, or can be obtained by the client by making an unauthorized protected resource request (from the "WWW-Authenticate" response header auth-uri (Section 6.1.2) attribute).

The authorization endpoint advertised by the resource server MAY include a query component as defined by [RFC3986] section 3.

Since requests to the authorization endpoint result in user authentication and the transmission of sensitive values, the authorization server SHOULD require the use of a transport-layer mechanism such as TLS/SSL (or a secure channel with equivalent protections) when sending requests to the authorization endpoints.

3.2. Token Endpoint

After obtaining authorization from the resource owner, clients request an access token from the authorization server's token endpoint.

The URI of the token endpoint can be found in the service documentation, or can be obtained by the client by making an unauthorized protected resource request (from the "WWW-Authenticate" response header token-uri (Section 6.1.2) attribute).

The token endpoint advertised by the resource server MAY include a query component as defined by [RFC3986] section 3.

Since requests to the token endpoint result in the transmission of plain text credentials in the HTTP request and response, the authorization server MUST require the use of a transport-layer mechanism such as TLS/SSL (or a secure channel with equivalent protections) when sending requests to the token endpoints.

3.2.1. Response Format

Authorization servers respond to client requests by including a set of response parameters in the entity body of the HTTP response. The response uses the "application/json" media type as defined by [RFC4627].

The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings. Numerical number are included as JSON numbers.

The authorization server MUST include the HTTP "Cache-Control" response header field with a value of "no-store" in any response containing tokens, secrets, or other sensitive information.

3.2.1.1. Access Token Response

After receiving and verifying a valid and authorized access token

request from the client (as described in each of the flows below), the authorization server constructs a JSON-formatted response which includes the common parameters set as well as additional flow-specific parameters. The formatted parameters are sent to the client in the entity body of the HTTP response with a 200 status code (OK).

The token response contains the following common parameters:

access_token
: REQUIRED. The access token issued by the authorization server.

expires_in
: OPTIONAL. The duration in seconds of the access token lifetime.

refresh_token
: OPTIONAL. The refresh token used to obtain new access tokens using the same end-user access grant as described in Section 4.

access_token_secret
: REQUIRED if requested by the client. The corresponding access token secret as requested by the client.

scope
: OPTIONAL. The scope of the access token as a list of space-delimited strings. The value of the "scope" parameter is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope.

For example (line breaks are for display purposes only):

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{"access_token":"SlAV32hkKG","expires_in":3600,
"refresh_token":"8xLOxBtZp8"}
```

3.2.1.2. Error Response

If the token request is invalid or unauthorized, the authorization server constructs a JSON-formatted response which includes the common parameters set as well as additional flow-specific parameters. The formatted parameters are sent to the client in the entity body of the HTTP response with a 400 status code (Bad Request).

The response contains the following common parameter:

error
: REQUIRED. The parameter value MUST be set to one of the values specified by each flow.

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store

{"error":"incorrect_client_credentials"}
```

3.3. Flow Parameters

The sizes of tokens and other values received from the authorization server, are left undefined by this specification. Clients should avoid making assumptions about value sizes. Servers should document the expected size of any value they issue.

Unless otherwise noted, all the protocol parameter names and values are case sensitive.

3.4. Client Credentials

When requesting access from the authorization server, the client identifies itself using a set of client credentials. The client credentials include a client identifier and an OPTIONAL symmetric

shared secret. The means through which the client obtains these credentials are beyond the scope of this specification, but usually involve registration with the authorization server.

The client identifier is used by the authorization server to establish the identity of the client for the purpose of presenting information to the resource owner prior to granting access, as well as for providing different service levels to different clients. They can also be used to block unauthorized clients from requesting access.

Due to the nature of some clients, authorization servers SHOULD NOT make assumptions about the confidentiality of client credentials without establishing trust with the client operator. Authorization servers SHOULD NOT issue client secrets to clients incapable of keeping their secrets confidential.

3.5. User-Agent Flow

The user-agent flow is a user delegation flow suitable for client applications residing in a user-agent, typically implemented in a browser using a scripting language such as JavaScript. These clients cannot keep client secrets confidential and the authentication of the

client is based on the user-agent's same-origin policy.

Unlike other flows in which the client makes separate authorization and access token requests, the client received the access token as a result of the authorization request in the form of an HTTP redirection. The client requests the authorization server to redirect the user-agent to another web server or local resource accessible to the browser which is capable of extracting the access token from the response and passing it to the client.

This user-agent flow does not utilize the client secret since the client executables reside on the end-user's computer or device which makes the client secret accessible and exploitable. Because the access token is encoded into the redirection URI, it may be exposed to the end-user and other applications residing on the computer or device.

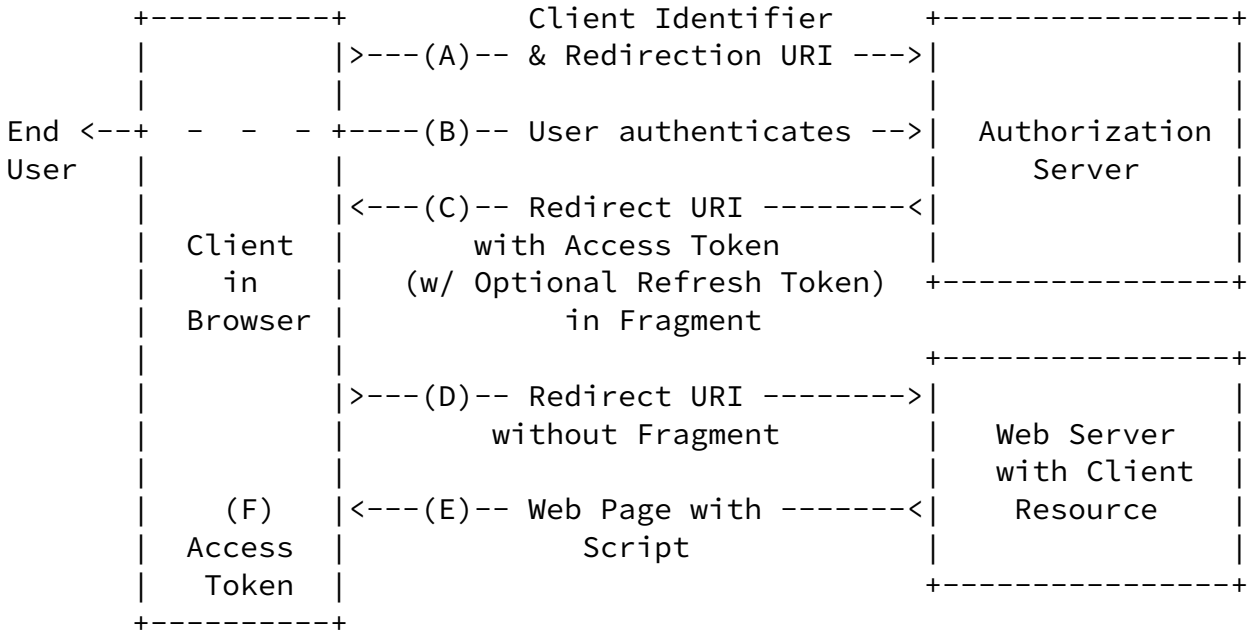


Figure 3

The user-agent flow illustrated in Figure 3 includes the following steps:

(A) The client sends the user-agent to the authorization server and includes its client identifier and redirection URI in the request.

(B) The authorization server authenticates the end-user (via the user-agent) and establishes whether the end-user grants or denies the client's access request.

(C) Assuming the end-user granted access, the authorization server redirects the user-agent to the redirection URI provided earlier. The redirection URI includes the access token in the URI fragment.

(D) The user-agent follows the redirection instructions by making a request to the web server which does not include the fragment.

The user-agent retains the fragment information locally.

(E) The web server returns a web page containing a script capable of extracting the access token from the URI fragment retained by the user-agent.

(F) The user-agent executes the script provided by the web server which extracts the access token and passes it to the client.

3.5.1. Client Requests Authorization

In order for the end-user to grant the client access, the client sends the end-user to the authorization server. The client constructs the request URI by adding the following URI query parameters to the user authorization endpoint URI:

type
: REQUIRED. The parameter value MUST be set to "user_agent".

client_id
: REQUIRED. The client identifier as described in Section 3.4.

redirect_uri
: REQUIRED unless a redirection URI has been established between the client and authorization server via other means. An absolute URI to which the authorization server will redirect the user-agent to when the end-user authorization step is completed. The authorization server SHOULD require the client to pre-register their redirection URI. Authorization servers MAY restrict the redirection URI to not include a query component as defined by [RFC3986] section 3.

state
: OPTIONAL. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client.

scope
: OPTIONAL. The scope of the access request expressed as a list of space-delimited strings. The value of the "scope" parameter is defined by the authorization server. If the value contains

multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope.

immediate
: OPTIONAL. The parameter value must be set to "true" or "false". If set to "true", the authorization server MUST NOT prompt the end-user to authenticate or approve access. Instead, the authorization server attempts to establish the end-user's identity via other means (e.g. browser cookies) and checks if the end-user has previously approved an identical access request by the same client and if that access grant is still active. If the authorization server does not support an immediate check or if it is unable to establish the end-user's identity or approval status, it MUST deny the request without prompting the end-user. Defaults to "false" if omitted.

secret_type
: OPTIONAL. The access token secret type as described by Section 5.3. If omitted, the authorization server will issue a bearer token (an access token without a matching secret) as described by Section 5.2.

The client directs the end-user to the constructed URI using an HTTP redirection response, or by other means available to it via the end-user's user-agent. The request MUST use the HTTP "GET" method.

For example, the client directs the end-user's user-agent to make the following HTTPS request (line breaks are for display purposes only):

```
  GET /authorize?type=user_agent&client_id=s6BhdRkqt3&
      redirect_uri=https%3A%2F%2FExample%2Ecom%2Frd HTTP/1.1
  Host: server.example.com
```

If the client has previously registered a redirection URI with the authorization server, the authorization server MUST verify that the redirection URI received matches the registered URI associated with the client identifier.

The authorization server authenticates the end-user and obtains an authorization decision (by asking the end-user or establishing approval via other means). The authorization server sends the end-

user's user-agent to the provided client redirection URI using an HTTP redirection response.

3.5.1.1. End-user Grants Authorization

If the end-user authorizes the access request, the authorization server issues an access token and delivers it to the client by adding the following parameters, using the "application/x-www-form-urlencoded" format as defined by [W3C.REC-html40-19980424], to the redirection URI fragment:

access_token
: REQUIRED. The access token.

expires_in
: OPTIONAL. The duration in seconds of the access token lifetime.

refresh_token
: OPTIONAL. The refresh token.

state
: REQUIRED if the "state" parameter was present in the client authorization request. Set to the exact value received from the client.

access_token_secret
: REQUIRED if requested by the client. The corresponding access token secret as requested by the client.

For example, the authorization server redirects the end-user's user-agent by sending the following HTTP response:

```
 HTTP/1.1 302 Found
 Location: http://example.com/rd#access_token=FJQbwq9&expires_in=3600
```

3.5.1.2. End-user Denies Authorization

If the end-user denied the access request, the authorization server responds to the client by adding the following parameters, using the "application/x-www-form-urlencoded" format as defined by [W3C.REC-html40-19980424], to the redirection URI fragment:

error
: REQUIRED. The parameter value MUST be set to "user_denied".

state
: REQUIRED if the "state" parameter was present in the client authorization request. Set to the exact value received from the client.

For example, the authorization server responds with the following:

```
HTTP/1.1 302 Found
Location: http://example.com/rd#error=user_denied
```

The authorization flow concludes unsuccessfully. To extract the error message, the client follows the steps described in Section 3.5.2.

3.5.2. Client Extracts Access Token

The user-agent follows the authorization server redirection response by making an HTTP "GET" request to the URI received in the "Location" HTTP response header. The user-agent SHALL NOT include the fragment component with the request.

For example, the user-agent makes the following HTTP "GET" request in response to the redirection directive received from the authorization server:

```
GET /rd HTTP/1.1
Host: example.com
```

The HTTP response to the redirection request returns a web page (typically an HTML page with an embedded script) capable of accessing the full redirection URI including the fragment retained by the user-agent, and extracting the access token (and other parameters) contained in the fragment.

3.6. Web Server Flow

The web server flow is a user delegation flow suitable for clients capable of interacting with the end-user's user-agent (typically a web browser) and capable of receiving incoming requests from the authorization server (capable of acting as an HTTP server).

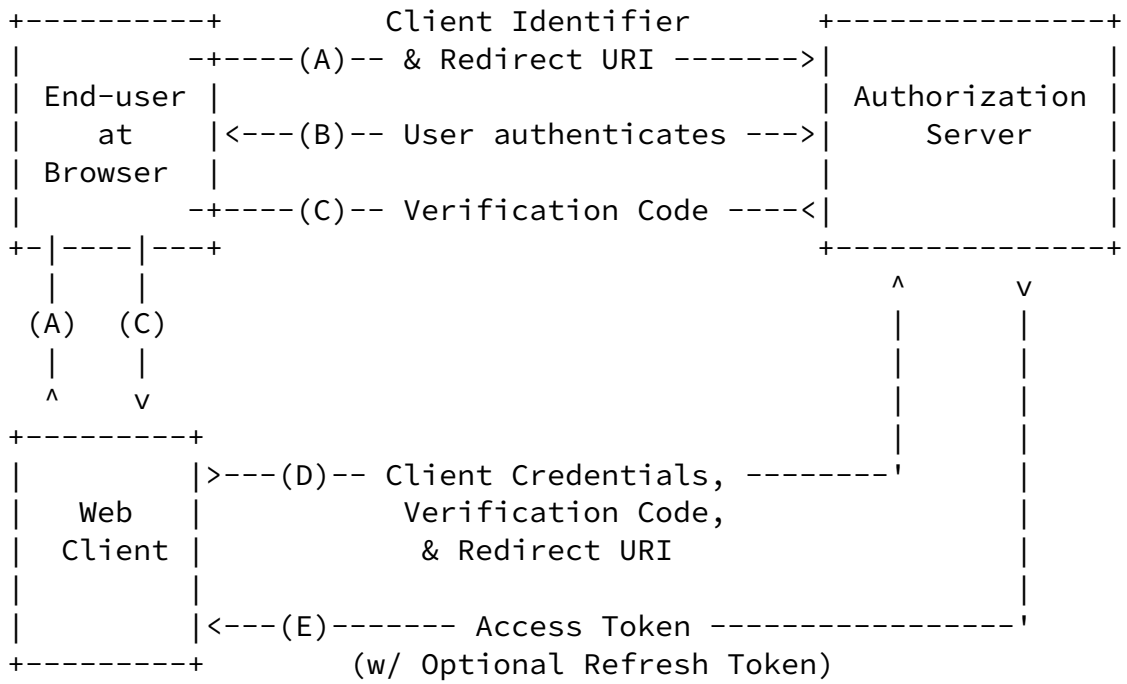


Figure 4

The web server flow illustrated in Figure 4 includes the following steps:

(A) The web client initiates the flow by redirecting the end-user's user-agent to the authorization endpoint with its client identifier and a redirect URI to which the authorization server will send the end-user back once authorization is received (or denied).

(B) The authorization server authenticates the end-user (via the user-agent) and establishes whether the end-user grants or

denies the client's access request.

(C) Assuming the end-user granted access, the authorization server redirects the user-agent back to the client to the redirection URI provided earlier. The authorization includes a verification code for the client to use to obtain an access token.

(D) The client requests an access token from the authorization server by including its client credentials (identifier and secret), as well as the verification code received in the previous step.

(E) The authorization server validates the client credentials and the verification code and responds back with the access token.

3.6.1. Client Requests Authorization

In order for the end-user to grant the client access, the client sends the end-user to the authorization server. The client constructs the request URI by adding the following URI query parameters to the user authorization endpoint URI:

type
: REQUIRED. The parameter value MUST be set to "web_server".

client_id
: REQUIRED. The client identifier as described in Section 3.4.

redirect_uri
: REQUIRED unless a redirection URI has been established between the client and authorization server via other means. An absolute URI to which the authorization server will redirect the user-agent to when the end-user authorization step is completed. The authorization server MAY require the client to pre-register their redirection URI. Authorization servers MAY restrict the redirection URI to not include a query component as defined by [RFC3986] section 3.

state
: OPTIONAL. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client.

scope
: OPTIONAL. The scope of the access request expressed as a list of space-delimited strings. The value of the "scope" parameter is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope.

immediate
: OPTIONAL. The parameter value must be set to "true" or "false". If set to "true", the authorization server MUST NOT prompt the end-user to authenticate or approve access. Instead, the authorization server attempts to establish the end-user's identity via other means (e.g. browser cookies) and checks if the end-user has previously approved an identical access request by the same client and if that access grant is

still active. If the authorization server does not support an immediate check or if it is unable to establish the end-user's identity or approval status, it MUST deny the request without prompting the end-user. Defaults to "false" if omitted.

The client directs the end-user to the constructed URI using an HTTP redirection response, or by other means available to it via the end-user's user-agent. The request MUST use the HTTP "GET" method.

For example, the client directs the end-user's user-agent to make the following HTTPS requests (line breaks are for display purposes only):

```
GET /authorize?type=web_server&client_id=s6BhdRkqt3&redirect_uri=
    https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1
Host: server.example.com
```

If the client has previously registered a redirection URI with the

authorization server, the authorization server MUST verify that the redirection URI received matches the registered URI associated with the client identifier.

The authorization server authenticates the end-user and obtains an authorization decision (by asking the end-user or establishing approval via other means). The authorization server sends the end-user's user-agent to the provided client redirection URI using an HTTP redirection response, or by other means available to it via the end-user's user-agent.

3.6.1.1. End-user Grants Authorization

If the end-user authorizes the access request, the authorization server generates a verification code and associates it with the client identifier and redirection URI. The authorization server constructs the request URI by adding the following parameters to the query component of redirection URI provided by the client:

code
: REQUIRED. The verification code generated by the authorization server.

state
: REQUIRED if the "state" parameter was present in the client authorization request. Set to the exact value received from the client.

The verification code should expire shortly after it is issued and

allowed for a single use.

For example, the authorization server redirects the end-user's user-agent by sending the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?code=i1WsRn1uB1
```

In turn, the end-user's user-agent makes the following HTTPS "GET" request:

```
  GET /cb?code=i1WsRn1uB1 HTTP/1.1
  Host: client.example.com
```

3.6.1.2. End-user Denies Authorization

If the end-user denied the access request, the authorization server constructs the request URI by adding the following parameters to the query component of the redirection URI provided by the client:

error
: REQUIRED. The parameter value MUST be set to "user_denied".

state
: REQUIRED if the "state" parameter was present in the client authorization request. Set to the exact value received from the client.

For example, the authorization server directs the client to make the following HTTP request:

```
  GET /cb?error=user_denied HTTP/1.1
  Host: client.example.com
```

The authorization flow concludes unsuccessfully.

3.6.2. Client Requests Access Token

The client obtains an access token from the authorization server by making an HTTP "POST" request to the token endpoint. The client constructs a request URI by adding the following parameters to the request:

type
: REQUIRED. The parameter value MUST be set to "web_server".

client_id
: REQUIRED. The client identifier as described in Section 3.4.

client_secret
: REQUIRED if the client identifier has a matching secret. The client secret as described in Section 3.4.

code
: REQUIRED. The verification code received from the authorization server.

redirect_uri
: REQUIRED. The redirection URI used in the initial request.

secret_type
: OPTIONAL. The access token secret type as described by Section 5.3. If omitted, the authorization server will issue a bearer token (an access token without a matching secret) as described by Section 5.2.

For example, the client makes the following HTTPS request (line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

type=web_server&client_id=s6BhdRkqt3&
client_secret=gX1fBat3bV&code=i1WsRn1uB1&
redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

The authorization server MUST verify that the verification code, client identity, client secret, and redirection URI are all valid and match its stored association. If the request is valid, the authorization server issues a successful response as described in Section 3.2.1.1.

For example (line breaks are for display purposes only):

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{"access_token":"SlAV32hkKG","expires_in":3600,
"refresh_token":"8xLOxBtZp8"}
```

If the request is invalid, the authorization server returns an error response as described in Section 3.2.1.2 with one of the following error codes:

- "redirect_uri_mismatch"
- "bad_verification_code"
- "incorrect_client_credentials"

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store

{"error":"incorrect_client_credentials"}
```

3.7. Device Flow

The device flow is a user delegation flow suitable for clients executing on devices which do not have an easy data-entry method (e.g. game consoles or media hub), but where the end-user has separate access to a user-agent on another computer or device (e.g. home computer, a laptop, or a smart phone). The client is incapable of receiving incoming requests from the authorization server (incapable of acting as an HTTP server).

Instead of interacting with the end-user's user-agent, the client instructs the end-user to use another computer or device and connect to the authorization server to approve the access request. Since the client cannot receive incoming requests, it polls the authorization server repeatedly until the end-user completes the approval process.

This device flow does not utilize the client secret since the client

executables reside on a local device which makes the client secret accessible and exploitable.

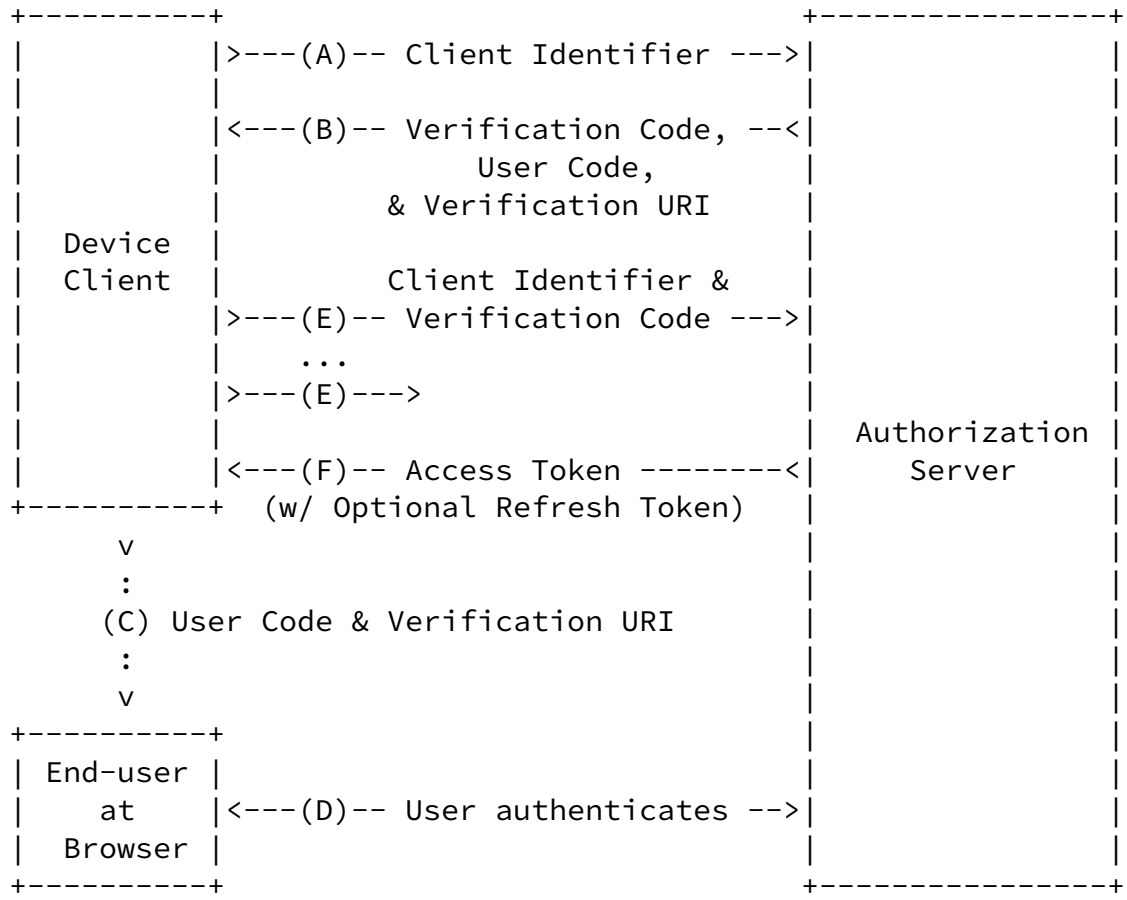


Figure 5

The device flow illustrated in Figure 5 includes the following steps:

(A) The client requests access from the authorization server and includes its client identifier in the request.

(B) The authorization server issues a verification code, a user code, and provides the end-user authorization URI.

(C) The client instructs the end-user to use its user-agent (elsewhere) and visit the provided authorization URI. The

client provides the user with the user code to enter in order to grant access.

(D) The authorization server authenticates the end-user (via the user-agent) and prompts the end-user to grant the client's access request. If the end-user agrees to the client's access request, the end-user enters the user code provided by the client.

(E) While the end-user authorizes (or denies) the client's request (D), the client repeatedly polls the authorization server to find out if the end-user completed the user authorization step. The client includes the verification code and its client identifier.

(F) Assuming the end-user granted access, the authorization server validates the verification code provided by the client and responds back with the access token.

3.7.1. Client Requests Authorization

The client initiates the flow by requesting a set of verification codes from the authorization server by making an HTTP "POST" request to the token endpoint. The client constructs a request URI by adding the following parameters to the request:

type
: REQUIRED. The parameter value MUST be set to "device_code".

client_id
: REQUIRED. The client identifier as described in Section 3.4.

scope
: OPTIONAL. The scope of the access request expressed as a list of space-delimited strings. The value of the "scope" parameter is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the

requested scope.

For example, the client makes the following HTTPS request (line breaks are for display purposes only):

```
POST /token?type=device_code&client_id=s6BhdRkqt3
      HTTP/1.1
Host: server.example.com
```

In response, the authorization server generates a verification code and a user code and includes them in the HTTP response body using the

"application/json" format as described by Section 3.2.1 with a 200 status code (OK). The response contains the following parameters:

code
: REQUIRED. The verification code.

user_code
: REQUIRED. The user code.

user_uri
: REQUIRED. The user authorization URI on the authorization server. The URI should be short and easy to remember as end-users will be asked to manually type it into their user-agent.

expires_in
: OPTIONAL. The duration in seconds of the verification code lifetime.

interval
: OPTIONAL. The minimum amount of time in seconds that the client SHOULD wait between polling requests to the token endpoint.

For example (line breaks are for display purposes only):

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
Cache-Control: no-store

{"code":"74tq5miHKB","user_code":"94248","user_uri":"http%3A%2F%2
Fwww%2Eexample%2Ecom%2Fdevice","interval"=5}
```

The client displays the user code and the user authorization URI to the end-user, and instructs the end-user to visit the URI using a user-agent and enter the user code.

The end-user manually types the provided URI and authenticates with the authorization server. The authorization server prompts the end-user to authorize the client's request by entering the user code provided by the client. Once the end-user approves or denies the request, the authorization server informs the end-user to return to the device for further instructions.

3.7.2. Client Requests Access Token

Since the client is unable to receive incoming requests from the authorization server, it polls the authorization server repeatedly until the end-user grants or denies the request, or the verification code expires.

The client makes the following request at an arbitrary but reasonable interval which MUST NOT exceed the minimum interval rate provided by the authorization server (if present via the "interval" parameter). Alternatively, the client MAY provide a user interface for the end-user to manually inform it when authorization was granted.

The client requests an access token by making an HTTP "POST" request to the token endpoint. The client constructs a request URI by adding the following parameters to the request:

type
: REQUIRED. The parameter value MUST be set to "device_token".

client_id

REQUIRED. The client identifier as described in Section 3.4.

code
: The verification code received from the authorization server.

secret_type
: OPTIONAL. The access token secret type as described by Section 5.3. If omitted, the authorization server will issue a bearer token (an access token without a matching secret) as described by Section 5.2.

For example, the client makes the following HTTPS request (line breaks are for display purposes only):

```
POST /token?type=device_token&client_id=s6BhdRkqt3
     &code=J2vC42OifV HTTP/1.1
Host: server.example.com
```

If the end-user authorized the request, the authorization server issues an access token response as described in Section 3.2.1.1.

For example (line breaks are for display purposes only):

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{"access_token":"SlAV32hkKG","expires_in":3600,
"refresh_token":"8xLOxBtZp8"}
```

If the request is invalid, the authorization server returns an error response as described in Section 3.2.1.2 with one of the following error codes:

- "authorization_declined"

- "bad_verification_code"

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store

{"error":"authorization_declined"}
```

If the end-user authorization is pending or expired without receiving any response from the end-user, or the client is exceeding the allowed polling interval, the authorization server returns an error response as described in Section 3.2.1.2 with one of the following error codes:

- "'authorization_pending"

- "slow_down"

- "code_expired"

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store
```

```
{"error":"authorization_pending"}
```

3.8. Username and Password Flow

The username and password flow is suitable for clients capable of asking end-users for their usernames and passwords. It is also used to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the end-user credentials stored with tokens.

However, unlike the HTTP Basic authentication scheme defined in [RFC2617], the end-user's credentials are used in a single request and are exchanged for an access token and refresh token which eliminates the client need to store them for future use.

The methods through which the client prompts end users for their usernames and passwords is beyond the scope of this specification. The client MUST discard the usernames and passwords once an access token has been obtained.

This flow is suitable in cases where the end-user already has a trust relationship with the client, such as its computer operating system or highly privileged applications. Authorization servers should take special care when enabling the username and password flow, and only when other delegation flows are not viable.

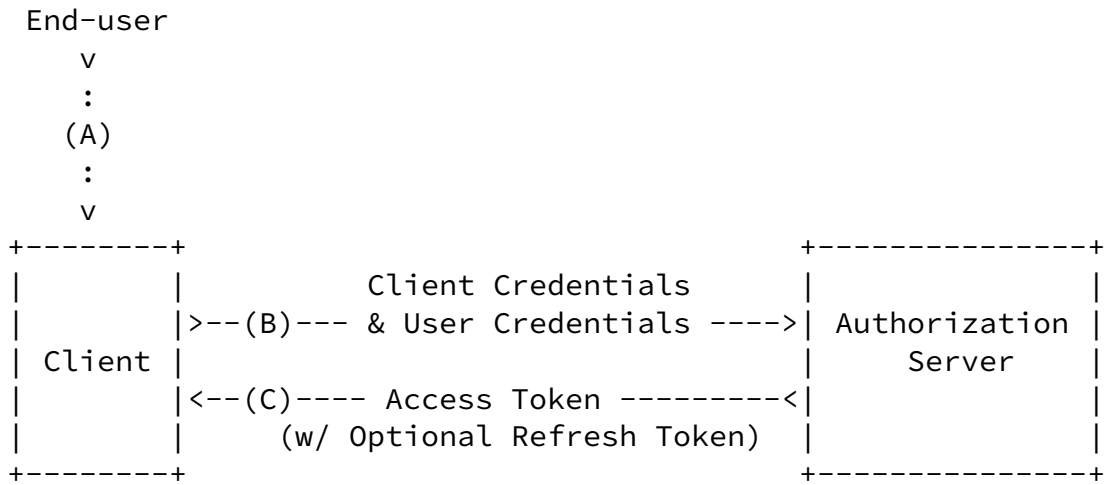


Figure 6

The username and password flow illustrated in Figure 6 includes the following steps:

(A) The end-user provides the client with its username and password.

(B) The client sends an access token request to the authorization server and includes its client identifier and client secret, and the end-user's username and password.

(C) The authorization server validates the end-user credentials and the client credentials and issues an access token.

3.8.1. Client Requests Access Token

The client requests an access token by making an HTTP "POST" request to the token endpoint. The client constructs a request URI by adding the following parameters to the request:

type
: REQUIRED. The parameter value MUST be set to "username".

client_id
: REQUIRED. The client identifier as described in Section 3.4.

client_secret
: REQUIRED. The client secret as described in Section 3.4. OPTIONAL if no client secret was issued.

username
: REQUIRED. The end-user's username.

password
: REQUIRED. The end-user's password.

scope
: OPTIONAL. The scope of the access request expressed as a list of space-delimited strings. The value of the "scope" parameter is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope.

secret_type
: OPTIONAL. The access token secret type as described by Section 5.3. If omitted, the authorization server will issue a bearer token (an access token without a matching secret) as

described by Section 5.2.

For example, the client makes the following HTTPS request (line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com

type=username&client_id=s6BhdRkqt3&client_secret=
47HDu8s&username=johndoe&password=A3ddj3w
```

The authorization server MUST validate the client credentials and end-user credentials and if valid issues an access token response as described in Section 3.2.1.1.

For example (line breaks are for display purposes only):

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{"access_token":"SlAV32hkKG","expires_in":3600,
"refresh_token":"8xLOxBtZp8"}
```

If the request is invalid, the authorization server returns an error response as described in Section 3.2.1.2 with one of the following error codes:

- "incorrect_client_credentials"
- "unauthorized_client'" - The client is not permitted to use this flow.

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store
```

```
{"error":"incorrect_client_credentials"}
```


3.9. Client Credentials Flow

The client credentials flow is used when the client acts on behalf of itself (the client is the resource owner), or when the client credentials are used to obtain an access token representing a previously established access authorization. The client secret is assumed to be high-entropy since it is not designed to be memorized by an end-user.

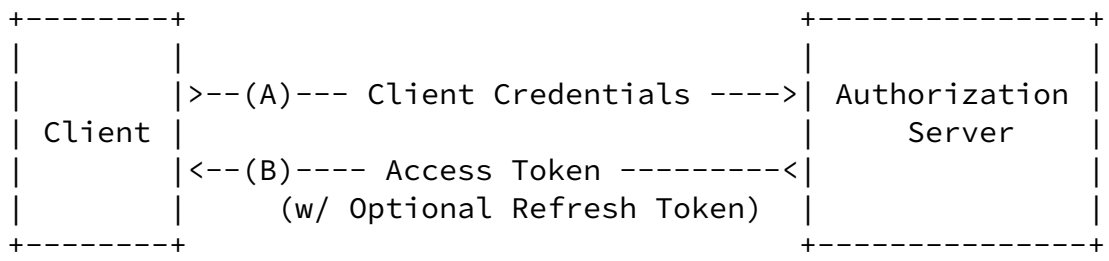


Figure 7

The client credential flow illustrated in Figure 7 includes the following steps:

(A) The client sends an access token request to the authorization server and includes its client identifier and client secret.

(B) The authorization server validates the client credentials and issues an access token.

3.9.1. Client Requests Access Token

The client requests an access token by making an HTTP "POST" request to the token endpoint. The client constructs a request URI by adding the following parameters to the request:

type
      REQUIRED. The parameter value MUST be set to
      "client_credentials".

client_id
      REQUIRED. The client identifier as described in Section 3.4.

client_secret
      REQUIRED. The client secret as described in Section 3.4.

scope
      OPTIONAL. The scope of the access request expressed as a list
      of space-delimited strings. The value of the "scope" parameter
      is defined by the authorization server. If the value contains
      multiple space-delimited strings, their order does not matter,
      and each string adds an additional access range to the
      requested scope.

secret_type
      OPTIONAL. The access token secret type as described by
      Section 5.3. If omitted, the authorization server will issue a
      bearer token (an access token without a matching secret) as
      described by Section 5.2.

For example, the client makes the following HTTPS request (line breaks are for display purposes only):

```
  POST /token HTTP/1.1
  Host: server.example.com

  type=client_credentials&client_id=s6BhdRkqt3&client_secret=47HDu8s
```

The authorization server MUST validate the client credentials and if valid issues an access token response as described in Section 3.2.1.1.

For example (line breaks are for display purposes only):

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{"access_token":"SlAV32hkKG","expires_in":3600,
"refresh_token":"8xLOxBtZp8"}
```

If the request is invalid, the authorization server returns an error response as described in Section 3.2.1.2 with one of the following error codes:

o "incorrect_client_credentials"

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store

{"error":"incorrect_client_credentials"}
```

3.10. Assertion Flow

The assertion flow is used when a client wishes to exchange an existing security token or assertion for an access token. This flow is suitable when the client is the resource owner or is acting on behalf of the resource owner (based on the content of the assertion used).

The assertion flow requires the client to obtain a assertion (such as a SAML [OASIS.saml-core-2.0-os] assertion) from an assertion issuer or to self-issue an assertion prior to initiating the flow. The assertion format, the process by which the assertion is obtained, and

the method of validating the assertion are defined by the assertion issuer and the authorization server, and are beyond the scope of this specification.

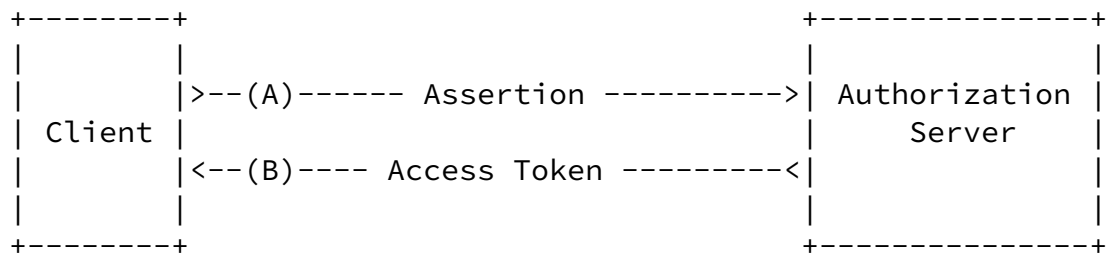


Figure 8

The assertion flow illustrated in Figure 8 includes the following steps:

(A) The client sends an access token request to the authorization server and includes an assertion.

(B) The authorization server validates the assertion and issues an access token.

3.10.1. Client Requests Access Token

The client requests an access token by making an HTTP "POST" request to the token endpoint. The client constructs a request URI by adding the following parameters to the request:

type
: REQUIRED. The parameter value MUST be set to "assertion".

format
: REQUIRED. The format of the assertion as defined by the authorization server. The value MUST be an absolute URI.

assertion
: REQUIRED. The assertion.

scope
: OPTIONAL. The scope of the access request expressed as a list of space-delimited strings. The value of the "scope" parameter is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope.

secret_type
: OPTIONAL. The access token secret type as described by Section 5.3. If omitted, the authorization server will issue a bearer token (an access token without a matching secret) as described by Section 5.2.

For example, the client makes the following HTTPS request (line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com

type=assertion&format=_______&assertion=_______
```

The authorization server MUST validate the assertion and if valid issues an access token response as described in Section 3.2.1.1. The authorization server SHOULD NOT issue a refresh token.

For example (line breaks are for display purposes only):

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{"access_token":"SlAV32hkKG","expires_in":3600}
```

If the request is invalid, the authorization server returns an error response as described in Section 3.2.1.2 with one of the following error codes:

o "invalid_assertion"

o "unknown_format"

For example:

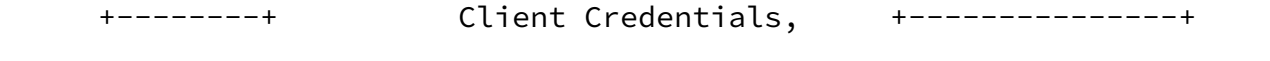
```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store

{"error":"invalid_assertion"}
```

Authorization servers SHOULD issue access tokens with a limited lifetime and require clients to refresh them by requesting a new access token using the same assertion if it is still valid. Otherwise the client MUST obtain a new valid assertion.

4. Refreshing an Access Token

Token refresh is used when the lifetime of an access token is shorter than the lifetime of the authorization grant. It allows clients to obtain a new access token without having to go through the authorization flow again or involve the resource owner. It is also used to obtain a new token with different security properties (e.g. bearer token, token with shared symmetric secret).



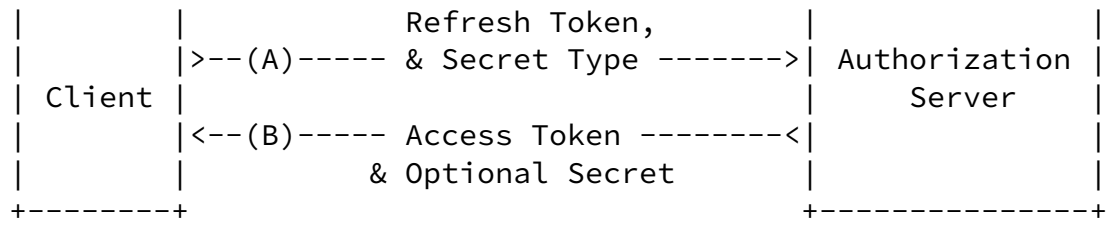


Figure 9

To refresh a token, the client constructs an HTTP "POST" request to the token endpoint and includes the following parameters in the HTTP request body using the "application/x-www-form-urlencoded" content type as defined by [W3C.REC-html40-19980424]:

type
: REQUIRED. The parameter value MUST be set to "refresh".

client_id
: REQUIRED. The client identifier as described in Section 3.4.

client_secret
: REQUIRED if the client was issued a secret. The client secret.

refresh_token
: REQUIRED. The refresh token associated with the access token to be refreshed.

secret_type
: OPTIONAL. The access token secret type as described by Section 5.3. If omitted, the authorization server will issue a bearer token (an access token without a matching secret) as described by Section 5.2.

For example, the client makes the following HTTPS request (line break are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

type=refresh_token&client_id=s6BhdRkqt3&client_secret=8eSEIpnqmM
&refresh_token=n4E9O119d&secret_type=hmac-sha256
```

verify the client credential, the validity of the refresh token, and

that the resource owner's authorization is still valid. If the request is valid, the authorization server issues an access token response as described in Section 3.2.1.1. The authorization server MAY issue a new refresh token in which case the client MUST NOT use the previous refresh token and replace it with the newly issued refresh token.

For example (line breaks are for display purposes only):

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{"access_token":"SlAV32hkKG","expires_in":3600}
```

If the request is invalid, the authorization server returns an error response as described in Section 3.2.1.2 with one of the following error codes:

- "incorrect_client_credentials"
- "authorization_expired"
- "unsupported_secret_type"

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store

{"error":"incorrect_client_credentials"}
```

5. Accessing a Protected Resource

Clients access protected resources by presenting an access token to the resource server. The methods used by the resource server to validate the access token are beyond the scope of this specification, but generally involve an interaction or coordination between the resource server and authorization server.

The method in which a client uses an access token depends on the security properties of the access tokens. By default, access tokens

are issued without a matching secret. Clients MAY request an access token with a matching secret by specifying the desired secret type using the "secret_type" token request parameter.

When an access token does not include a matching secret, the access token acts as a bearer token, where the token string is a shared symmetric secret. This requires treating the access token with the same care as other secrets (e.g. user passwords). Access tokens SHOULD NOT be sent in the clear over an insecure channel.

However, when it is necessary to transmit bearer tokens in the clear without a secure channel, authorization servers SHOULD issue access tokens with limited scope and lifetime to reduce the potential risk from a compromised access token. Clients SHOULD request and utilize an access token with a matching secret when making protected resource requests over an insecure channel (e.g. an HTTP request without using TLS/SSL).

When an access token includes a matching secret, the secret is not included directly in the request but is used instead to generate a cryptographic signature of the request. The signature can only be generated and verified by entities with access to the secret.

Clients SHOULD NOT make authenticated requests with an access token to unfamiliar resource servers, especially when using bearer tokens, regardless of the presence of a secure channel.

5.1. The Authorization Request Header

The "Authorization" request header field is used by clients to make both bearer token and cryptographic token requests. When making bearer token requests, the client uses the "token" attribute to include the access token in the request without any of the other attributes. Additional methods for making bearer token requests are described in Section 5.2.

For example:

```
GET /resource HTTP/1.1
Host: server.example.com
Authorization: Token token="vF9dft4qmT"
```

When making a cryptographic token request (using an access token with a matching secret) the client uses the "token" attribute to include the access token in the request, and uses the "nonce", "timestamp", "algorithm", and "signature" attributes to apply the matching secret.

For example:

```
GET /resource HTTP/1.1
Host: server.example.com
Authorization: Token token="vF9dft4qmT",
                     nonce="s8djwd",
                     timestamp="137131200",
                     algorithm="hmac-sha256",
                     signature="wOJIO9A2W5mFwDgiDvZbTSMK/PY="
```

The "Authorization" header field uses the framework defined by [RFC2617] as follows:

```
credentials    = "Token" RWS token-response

token-response = token-id
                 [ CS nonce ]
                 [ CS timestamp ]
                 [ CS algorithm ]
                 [ CS signature ]

token-id         = "token" "=" <"> token <">
timestamp        = "timestamp" "=" <"> 1*DIGIT <">
nonce            = "nonce" "=" <"> token <">

algorithm        = "algorithm" "=" algorithm-name
algorithm-name   = "hmac-sha256" /
                   token

signature        = "signature" "=" <"> token <">
```

5.2. Bearer Token Requests

Clients make bearer token requests by including the access token using the HTTP "Authorization" request header with the "Token" authentication scheme as described in Section 5.1. The access token is included using the "token" parameter.

For example, the client makes the following HTTPS request:

```
GET /resource HTTP/1.1
Host: server.example.com
Authorization: Token token="vF9dft4qmT"
```


The resource server MUST validate the access token and ensure it has not expired and that its scope covers the requested resource. If the token expired or is invalid, the resource server MUST reply with an HTTP 401 status code (Unauthorized) and include the HTTP "WWW-Authenticate" response header as described in Section 6.1.

For example:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Token realm='Service', error='token_expired'
```

Alternatively, the client MAY include the access token using the HTTP request URI in the query component as described in Section 5.2.1, or in the HTTP body when using the "application/x-www-form-urlencoded" content type as described in Section 5.2.2. Clients SHOULD only use the request URI or body when the "Authorization" request header is not available, and MUST NOT use more than one method in each request.

5.2.1. URI Query Parameter

When including the access token in the HTTP request URI, the client adds the access token to the request URI query component as defined by [RFC3986] using the "oauth_token" parameter.

For example, the client makes the following HTTPS request:

```
GET /resource?oauth_token=vF9dft4qmT HTTP/1.1
Host: server.example.com
```

The HTTP request URI query can include other request-specific parameters, in which case, the "oauth_token" parameters SHOULD be appended following the request-specific parameters, properly separated by an "&" character (ASCII code 38).

The resource server MUST validate the access token and ensure it has not expired and its scope includes the requested resource. If the resource expired or is not valid, the resource server MUST reply with an HTTP 401 status code (Unauthorized) and include the HTTP "WWW-Authenticate" response header as described in Section 6.1.

5.2.2. Form-Encoded Body Parameter

When including the access token in the HTTP request entity-body, the client adds the access token to the request body using the

"oauth_token" parameter. The client can use this method only if the following REQUIRED conditions are met:

- The entity-body is single-part.
- The entity-body follows the encoding requirements of the "application/x-www-form-urlencoded" content-type as defined by [W3C.REC-html40-19980424].
- The HTTP request entity-header includes the "Content-Type" header field set to "application/x-www-form-urlencoded".
- The HTTP request method is "POST", "PUT", or "DELETE".

The entity-body can include other request-specific parameters, in which case, the "oauth_token" parameters SHOULD be appended following the request-specific parameters, properly separated by an "&" character (ASCII code 38).

For example, the client makes the following HTTPS request:

```
POST /resource HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

oauth_token=vF9dft4qmT
```

The resource server MUST validate the access token and ensure it has not expired and its scope includes the requested resource. If the resource expired or is not valid, the resource server MUST reply with an HTTP 401 status code (Unauthorized) and include the HTTP "WWW-Authenticate" response header as described in Section 6.1.

5.3. Cryptographic Tokens Requests

Clients make authenticated protected resource requests using an access token with a matching secret by calculating a set of values and including them in the request using the "Authorization" header field. The way clients calculate these values depends on the access token secret type as issued by the authorization server.

This specification defines the "hmac-sha256" algorithm, and establishes a registry for providing additional algorithms. Clients obtain an access token with a matching "hmac-sha256" secret by using the "secret_type" parameter when requesting an access token.

5.3.1. The 'hmac-sha256' Algorithm

The "hmac-sha256" algorithm uses the HMAC method as defined in [RFC2104] together with the SHA-256 hash function defined in [NIST FIPS-180-3] to apply the access token secret to the request and generate a signature value that is included in the request instead of transmitting the secret in the clear.

To use the "hmac-sha256" algorithm, clients:

1. Calculate the request timestamp and generate a request nonce as

described in Section 5.3.1.1.

2. Construct the normalized request string as described in Section 5.3.1.2.

3. Calculate the request signature as described in Section 5.3.1.3.

4. Include the timestamp, nonce, algorithm name, and calculated signature in the request using the "Authorization" header field.

For example:

```
GET /resource HTTP/1.1
Host: server.example.com
Authorization: Token token="vF9dft4qmT",
                     nonce="s8djwd",
                     timestamp="137131200",
                     algorithm="hmac-sha256",
                     signature="wOJIO9A2W5mFwDgiDvZbTSMK/PY="
```

The resource server MUST validate the access token and ensure it has not expired and that its scope covers the requested resource. The resource server MUST also recalculate the request signature using the attributes provided by the client and compare it to the signature provided. If the token expired or is invalid, or if the signature is incorrect, the resource server MUST reply with an HTTP 401 status code (Unauthorized) and include the HTTP "WWW-Authenticate" response header as described in Section 6.1.

For example:

```
HTTP/1.1 401 Unauthorized
```

```
Date: Tue, 15 Nov 2010 08:12:31 GMT
WWW-Authenticate: Token realm='Service',
                        algorithms='hmac-sha256',
                        error='invalid_signature'
```

[[Errors list]]

5.3.1.1. Nonce and Timestamp

A timestamp in combination with unique nonce values is used to protect against replay attacks when transmitted over an insecure channel.

The nonce is a random string, uniquely generated by the client to allow the resource server to verify that a request has never been made before and helps prevent replay attacks when requests are made over a non-secure channel. The nonce value MUST be unique across all requests with the same timestamp and token combinations.

The timestamp value is the current time expressed in the number of seconds since January 1, 1970 00:00:00 GMT, and MUST be a positive integer.

To avoid the need to retain an infinite number of nonce values for future checks, resource servers MAY choose to restrict the time period after which a request with an old timestamp is rejected. When resource servers apply such a restriction, clients SHOULD synchronize their clocks by using the resource server's time as indicated by the HTTP "Date" response header field as defined in [RFC2616].

5.3.1.2. Normalized String Construction

The normalized request string is a consistent, reproducible concatenation of several of the HTTP request elements into a single string. The string is used as an input to the selected cryptographic method and includes the HTTP request method (e.g. "GET", "POST", etc.), the authority as declared by the HTTP "Host" request header, and the request resource URI.

The normalized request string does not cover the entire HTTP request. Most notably, it does not include the entity-body or most HTTP entity-headers. It is important to note that the resource server cannot verify the authenticity of the excluded request elements

without using additional protections such as TLS/SSL.

The normalized request string is constructed by concatenating together, in order, the following HTTP request elements, separated by the "," character (ASCII code 44):

1. The request timestamp as described in Section 5.3.1.1.

2. The request nonce as described in Section 5.3.1.1.

3. The cryptographic algorithm used.

4. The HTTP request method in uppercase. For example: "HEAD", "GET", "POST", etc.

5. The hostname, colon-separated (ASCII code 58) from the TCP port used to make the request as included in the HTTP request "Host" header field. The port MUST be included even if it is not included in the "Host" header field (i.e. the default port for the scheme).

6. The request resource URI.

For example, the normalized request string for the "GET" request URI "http://example.com/resource", request timestamp "137131200", request nonce "s8djwd", and "hmac-sha256" algorithm (line breaks are for display purposes only):

```
137131200,s8djwd,hmac-sha256,GET,example.com:80,
http://example.com/resource
```

5.3.1.3. Signature Calculation

Clients calculate the request signature using the HMAC-SHA256 function:

```
digest = HMAC-SHA256 (key, text)
```

by setting the function variables are follows:

text
: is set to the value of the normalize request string as described in Section 5.3.1.2.

key
: is set to the access token secret.

The request signature is the calculated value of the "digest" variable after the result octet string is base64-encoded per [RFC2045] section 6.8.

6. Identifying a Protected Resource

Clients access protected resources after locating the appropriate authorization and token endpoints and obtaining an access token. In many cases, interacting with a protected resource requires prior knowledge of the protected resource properties and methods, as well as its authentication requirements (i.e. establishing client identity, locating the authorization and token endpoints).

However, there are cases in which clients are unfamiliar with the protected resource, including whether the resource requires authentication. When clients attempt to access an unfamiliar protected resource without an access token, the resource server denies the request and informs the client of the required credentials using an HTTP authentication challenge.

In addition, when receiving an invalid authenticated request, the resource server issues an authentication challenge including the error type and message.

6.1. The WWW-Authenticate Response Header

A resource server receiving a request for a protected resource without a valid access token MUST respond with a 401 HTTP status code (Unauthorized), and includes at least one "Token" "WWW-Authenticate" response header field challenge.

The "WWW-Authenticate" header field uses the framework defined by [RFC2617] as follows:

```
challenge       = "Token" RWS token-challenge

token-challenge = realm
                  [ CS authz-uri ]
                  [ CS token-uri ]
                  [ CS algorithms ]
                  [ CS error ]

authz-uri       = "auth-uri" "=" URI-Reference
token-uri       = "token-uri" "=" URI-Reference
algorithms      = "algorithms" "=" <"> 1#algorithm-name  <">
error           = "error" "=" <"> token <">

CS              = OWS "," OWS
```

6.1.1. The 'realm' Attribute

The "realm" attribute is used to provide the protected resources partition as defined by [RFC2617].

6.1.2. The 'authorization-uri' Attribute

6.1.3. The 'algorithms' Attribute

6.1.4. The 'error' Attribute

7. Security Considerations

[[Todo]]

8. IANA Considerations

[[Not Yet]]

9. Acknowledgements

[[Add OAuth 1.0a authors + WG contributors]]

Appendix A. Differences from OAuth 1.0a

[[Todo]]

Appendix B. Document History

[[to be removed by RFC editor before publication as an RFC]]

-04

- Changed all token endpoints to use "POST"
- Clarified the authorization server's ability to issue a new refresh token when refreshing a token.
- Changed the flow categories to clarify the autonomous group.
- Changed client credentials language not to always be server-issued.
- Added a "scope" response parameter.
- Fixed typos.
- Fixed broken document structure.

-03

- Fixed typo in JSON error examples.

- Fixed general typos.
- Moved all flows sections up one level.

-02

- Removed restriction on "redirect_uri" including a query.
- Added "scope" parameter.
- Initial proposal for a JSON-based token response format.

-01

- Editorial changes based on feedback from Brian Eaton, Bill Keenan, and Chuck Mortimore.
- Changed device flow "type" parameter values and switch to use only the token endpoint.

-00

- Initial draft based on a combination of WRAP and OAuth 1.0a.

10. References

10.1. Normative References

[I-D.ietf-httpbis-p1-messaging]
Fielding, R., Gettys, J., Mogul, J., Nielsen, H., Masinter, L., Leach, P., Berners-Lee, T., and J. Reschke, "HTTP/1.1, part 1: URIs, Connections, and Message Parsing", draft-ietf-httpbis-p1-messaging-09 (work in progress), March 2010.

[NIST FIPS-180-3]
National Institute of Standards and Technology, "Secure Hash Standard (SHS). FIPS PUB 180-3, October 2008".

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail

Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform

Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[W3C.REC-html40-19980424]
Hors, A., Raggett, D., and I. Jacobs, "HTML 4.0 Specification", World Wide Web Consortium Recommendation REC-html40-19980424, April 1998, <http://www.w3.org/TR/1998/REC-html40-19980424>.

10.2. Informative References

[I-D.hammer-oauth]
Hammer-Lahav, E., "The OAuth 1.0 Protocol", draft-hammer-oauth-10 (work in progress), February 2010.

[I-D.hardt-oauth]
Hardt, D., Tom, A., Eaton, B., and Y. Goland, "OAuth Web Resource Authorization Profiles", draft-hardt-oauth-01 (work in progress), January 2010.

[OASIS.saml-core-2.0-os]
Cantor, S., Kemp, J., Philpott, R., and E. Maler, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-core-2.0-os, March 2005.

Authors' Addresses

Eran Hammer-Lahav (editor)
Yahoo!

Email: eran@hueniverse.com
URI: http://hueniverse.com

David Recordon
Facebook

Email: davidrecordon@facebook.com
URI: http://www.davidrecordon.com/

Dick Hardt

Email: dick.hardt@gmail.com
URI: http://dickhardt.org/