Network Working Group
Internet-Draft
Obsoletes: 5849 (if approved)
Intended status: Standards Track
Expires: June 4, 2011

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

December 1, 2010

The OAuth 2.0 Protocol Framework
draft-ietf-oauth-v2-11

Abstract

This specification describes the OAuth 2.0 protocol framework.

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 June 4, 2011.

Copyright Notice

Table of Contents

1. 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, the client accesses a protected resource on the server by authenticating with the server using the resource owner's credentials. In order to provide third-party applications access to protected resources, the resource owner shares its credentials with the third-party. This creates several problems and limitations:

- Third-party applications are required to store the resource-owner's credentials for future use, typically a password in clear-text.
- Servers are required to support password authentication, despite the security weaknesses created by passwords.
- Third-party applications gain overly broad access to the resource-owner's protected resources, leaving resource owners without any ability to restrict access to a limited subset of resources, to limit access duration, or to limit access to the methods supported by these resources.
- Resource owners cannot revoke access to an individual third-party

without revoking access to all third-parties, and must do so by changing their password.

OAuth addresses these issues by separating the role of the client from that of the resource owner. In OAuth, the client (which is usually not the resource owner, but is acting on the resource owner's behalf) requests access to resources controlled by the resource owner and hosted by the resource server, and is issued a different set of credentials than those of the resource owner.

Instead of using the resource owner's credentials to access protected resources, the client obtains an access token - a string which denotes a specific scope, duration, and other attributes. Access tokens are issued to third-party clients by an authorization server with the approval of the resource owner. The client uses the access token to access the protected resources hosted by the resource server.

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 an authentication service trusted by the photo sharing service (authorization server) which issues the printing service delegation-specific credentials (token).

Access tokens can have different formats, structures, and methods of utilization (e.g. cryptographic properties), based on the resource server security requirements. Access token attributes and the methods used to access protected resources are beyond the scope of this specification and are defined by companion specifications. The interaction between the authorization server and resource server is beyond the scope of this specification.

1.1. 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 following rules are included from [RFC3986]: URI-reference; and from [I-D.ietf-httpbis-p1-messaging]: OWS, RWS, and quoted-string.

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

1.2. Terminology

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

resource server
: A server capable of accepting and responding to protected resource requests.

client
: An application obtaining authorization and making protected resource requests.

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

end-user
: A human resource owner.

token
: A string representing an access authorization issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization servers. The token may denote an identifier used to retrieve the authorization information, or self-contain the authorization information in a verifiable manner (i.e. a token string consisting of some data and a signature). Tokens may be pure capabilities. Specific additional authentication credentials may be required in order for a client to use a token.

access token
: A token used by the client to make authenticated requests on behalf of the resource owner.

refresh token
: A token used by the client to obtain a new access token without having to involve the resource owner.

authorization code
: A short-lived token representing the authorization provided by the end-user. The authorization code is used to obtain an access token and a refresh token.

access grant
: A general term used to describe the intermediate credentials (such as an end-user password or authorization code) representing the resource owner authorization. Access grants are used by the client to obtain an access token. By exchanging access grants of different types for an access token, the resource server is only required to support a single authentication scheme.

authorization server
: A 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. A single authorization server may issue tokens for multiple resource servers.

end-user authorization endpoint
: The authorization server's HTTP endpoint capable of authenticating the end-user and obtaining authorization. The end-user authorization endpoint is described in Section 4.

token endpoint
: The authorization server's HTTP endpoint capable of issuing tokens and refreshing expired tokens. The token endpoint is described in Section 5.

client identifier
: A unique identifier issued to the client to identify itself to the authorization server. Client identifiers may have a

matching secret. The client identifier is described in Section 3.

1.3. Overview

OAuth provides a method for clients to access a protected resource on behalf of a resource owner. Before a client can access a protected resource, it must first obtain authorization (access grant) from the resource owner, then exchange the access grant for an access token (representing the grant's scope, duration, and other attributes). The client accesses the protected resource by presenting the access token to the resource server.

The access token provides an abstraction layer, replacing different authorization constructs (e.g. username and password, assertion) for a single token understood by the resource server. This abstraction enables issuing access tokens valid for a short time period, as well as removing the resource server's need to understand a wide range of authentication schemes.

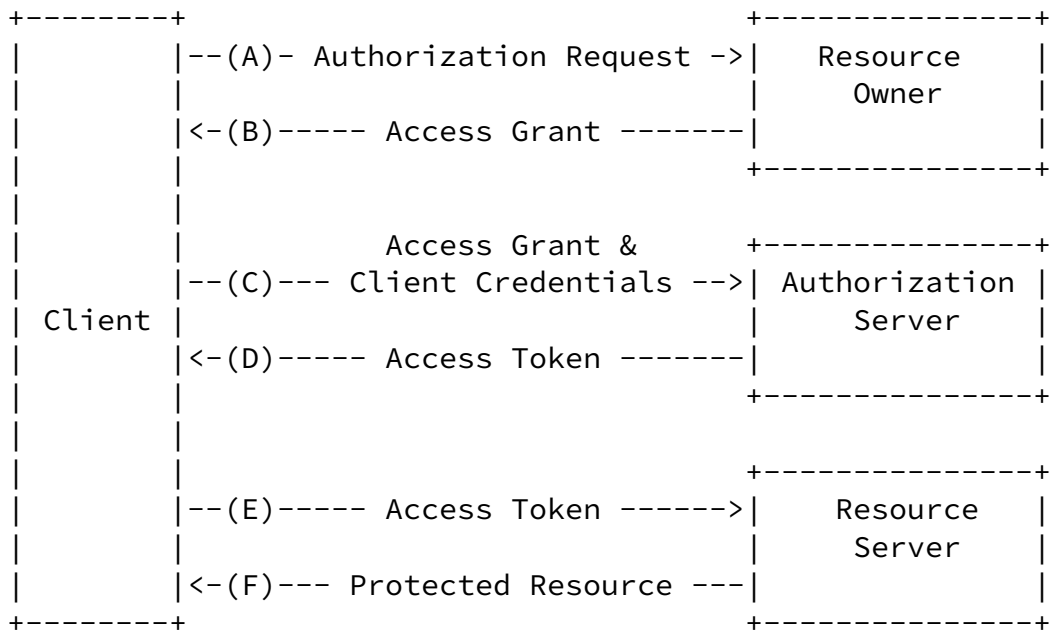


Figure 1: Abstract Protocol Flow

The abstract flow illustrated in Figure 1 describes the overall protocol architecture and includes the following steps:

(A) The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner, or preferably indirectly via an intermediary such as an authorization server.

(B) The client receives an access grant which represents the authorization provided by the resource owner.

(C) The client requests an access token by authenticating with the authorization server using its client credentials, and presenting the access grant.

(D) The authorization server validates the client credentials and the access grant, and if valid issues an access token.

(E) The client makes a protected resource request to the resource server by presenting the access token.

(F) The resource server validates the access token, and if valid, serves the request.

1.4. Access Grants

The access grant represents the authorization provided by the resource owner. The access grant type depends on the method used by the client and supported by the authorization server to obtain it.

1.4.1. Authorization Code

The authorization code is an access grant obtained by directing the end-user to an authorization server. The authorization server authenticates the end-user, obtains authorization, and issues the an authorization code to the client. Because the end-user only authenticates with the authorization server, the end-user's password is never shared with the client.

The authorization code access grant is suitable when the client is interacting with an end-user via a user-agent.

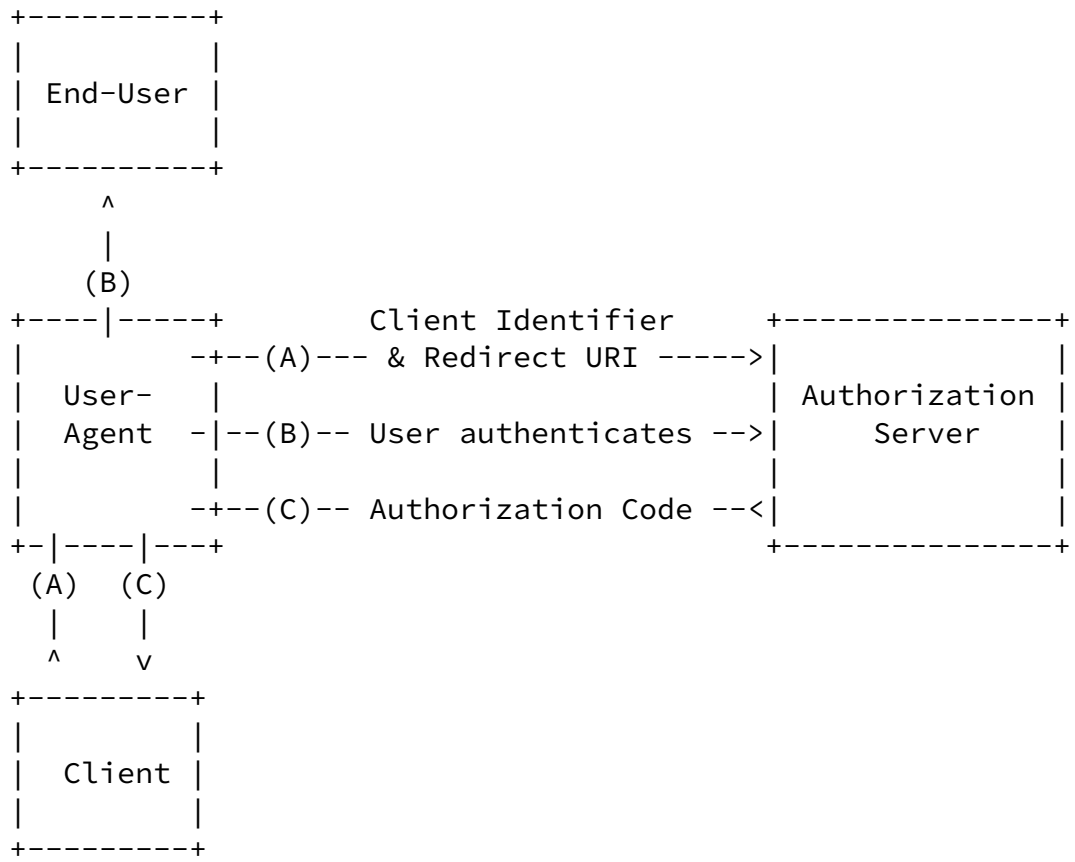


Figure 2: Obtaining an Authorization Code

The authorization code flow illustrated in Figure 2 includes the following steps:

(A) The client initiates the flow by directing the end-user's user-agent to the authorization server's end-user authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI (to which the authorization server will send the user-agent back once access is granted 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) If access is granted, the authorization server directs the user-agent back to the client using the redirection URI provided. The authorization server includes an authorization code for the client to use to obtain an access token.

Once the client obtains an authorization code, it requests an access token by authenticating with the authorization server (using its

client credentials) and presenting the authorization code (access grant).

In cases where the client is incapable of maintaining its client credentials secret (such as native applications or an application implemented as a user-agent script), the authorization server issues an access token directly to the client in step (C), instead of issuing an authorization code.

Obtaining an authorization code is described in Section 4.

1.4.2. Resource Owner Password Credentials

The resource owner password credentials (e.g. a username and password) can be used directly as an access grant to obtain an access token. The credentials should only be used when there is a high degree of trust between the resource owner and the client (e.g. its computer operating system or a highly privileged application), and when other access grant types are not available (such as an authorization code).

Even though this grant type requires direct client access to the resource owner's credentials, the resource owner's credentials are used for a single request and are exchanged for an access token. Unlike the HTTP Basic authentication scheme defined in [RFC2617], this grant type eliminates the need for the client to store the resource-owner's credentials for future use.

In Figure 3, the client requests authorization from the resource owner directly. When the resource owner is an end-user, the client typically prompts the end-user for the username and password.

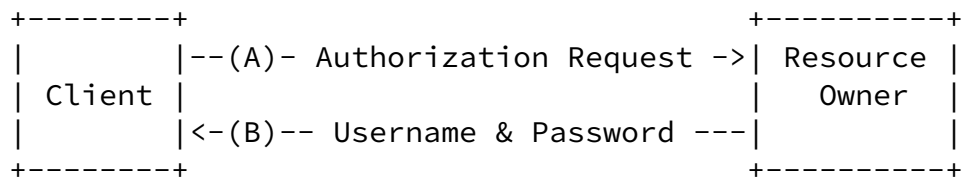


Figure 3: Obtaining Resource Owner Password Credentials

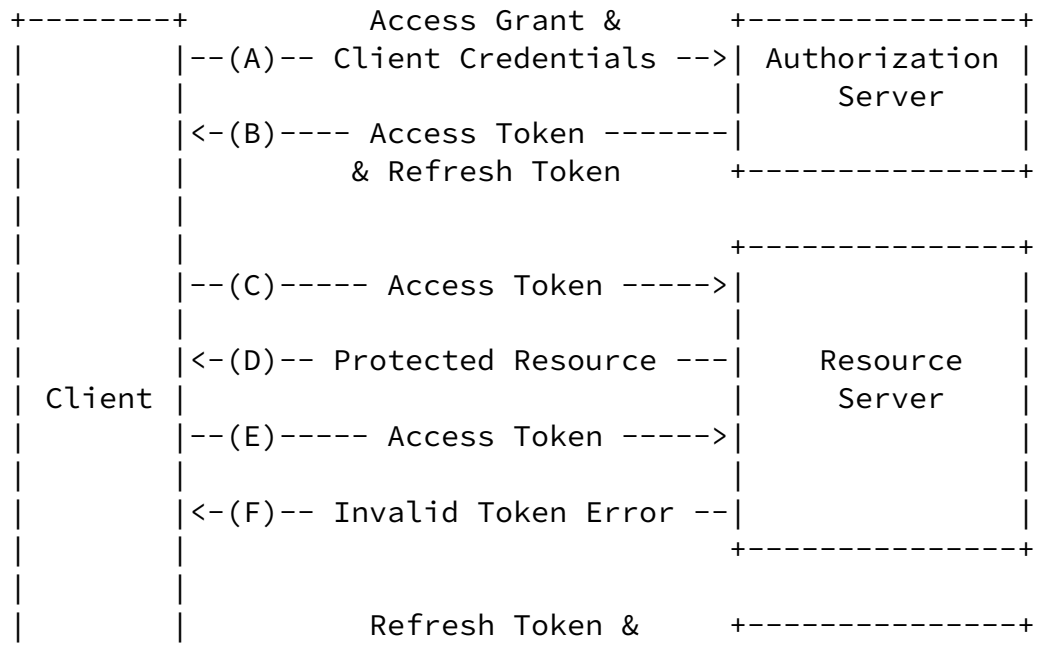
1.4.3. Client Credentials

The client credentials can be used as an access grant when the authorization scope is limited to the protected resources under the control of the client, or other protected resources previously arranged with the authorization server. Client credentials are used

as an access grant typically when the client is acting on its own behalf (the client is also the resource owner).

1.4.4. Refresh Token

Access tokens usually have a shorter lifetime than authorized by the resource owner. When issuing an access token, the authorization server can include a refresh token which is used by the client to obtain a new access token when the current access token expires. When requesting a new access token, the refresh token acts as an access grant. Using a refresh token removes the need to interact with the resource owner again, or to store the original access grant used to obtain the access token and refresh token.



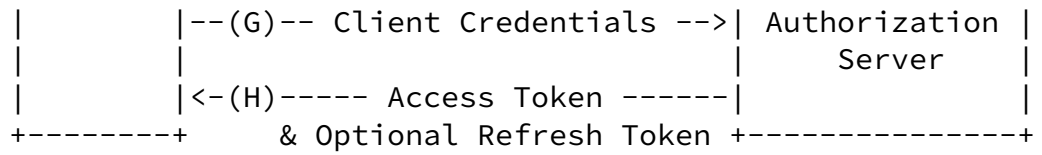


Figure 4: Refreshing an Access Token

The refresh token flow illustrated in Figure 4 includes the following steps:

(A) The client requests an access token by authenticating with the authorization server using its client credentials, and presenting an access grant.

(B) The authorization server validates the client credentials and the access grant, and if valid issues an access token and a refresh token.

(C) The client makes a protected resource requests to the resource server by presenting the access token.

(D) The resource server validates the access token, and if valid, serves the request.

(E) Steps (C) and (D) repeat until the access token expires. If the client does not know the access token expired, it makes another protected resource request. Otherwise, it skips to step (G).

(F) Since the access token is invalid (expired), the resource server returns an invalid token error.

(G) The client requests a new access token by authenticating with the authorization server using its client credentials, and presenting the refresh token (as the access grant).

(H) The authorization server validates the client credentials and the refresh token, and if valid issues a new access token (and optionally, a new refresh token).

1.4.5. Assertion

Assertions provide a bridge between OAuth and other trust frameworks. They enable the client to utilize existing trust relationships in order to obtain an access token. The access grant represented by an assertion depends on the assertion type, its content, and how it was issued, which are beyond the scope of this specification.

Assertions are used as part of the protocol extensibility model, providing a way for authorization servers to support additional access grant types.

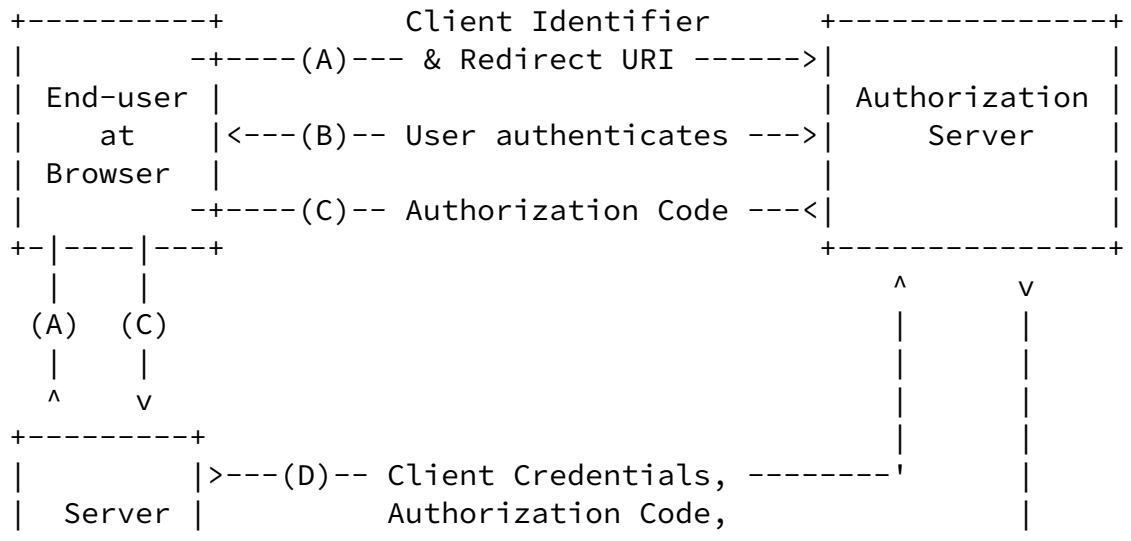
2. Client Profiles

[[add intro and find new names for the profiles. this section will have normative language in future drafts, similar to -05 and earlier.]]

2.1. Web Server

The web server profile is suitable for clients capable of interacting with the end-user's user-agent (typically a web browser) and capable

of receiving incoming requests (via redirection) from the authorization server (capable of acting as an HTTP server).



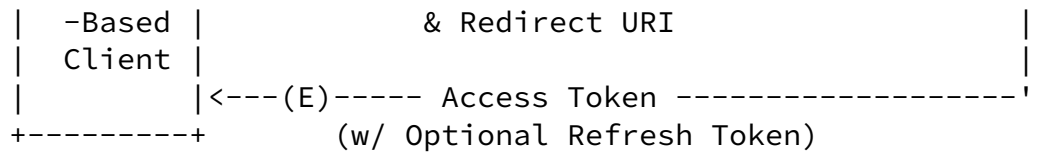


Figure 5: Web Server Flow

The web server flow illustrated in Figure 5 includes the following steps:

(A) The web client initiates the flow by redirecting the end-user's user-agent to the end-user authorization endpoint as described in Section 4. The client includes its client identifier, requested scope, local state, and a redirect URI to which the authorization server will send the end-user back once access is granted (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 an authorization code for the client to use to obtain an access token.

(D) The client requests an access token from the authorization server by authenticating and including the authorization code received in the previous step as described in Section 5.

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

2.2. User-Agent

The user-agent profile is 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 profiles in which the client makes separate requests for end-user authorization and access token, the client receives the access token as a result of the end-user 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 user-agent which is capable of extracting the access token from the response and passing it to the client.

This user-agent profile 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.

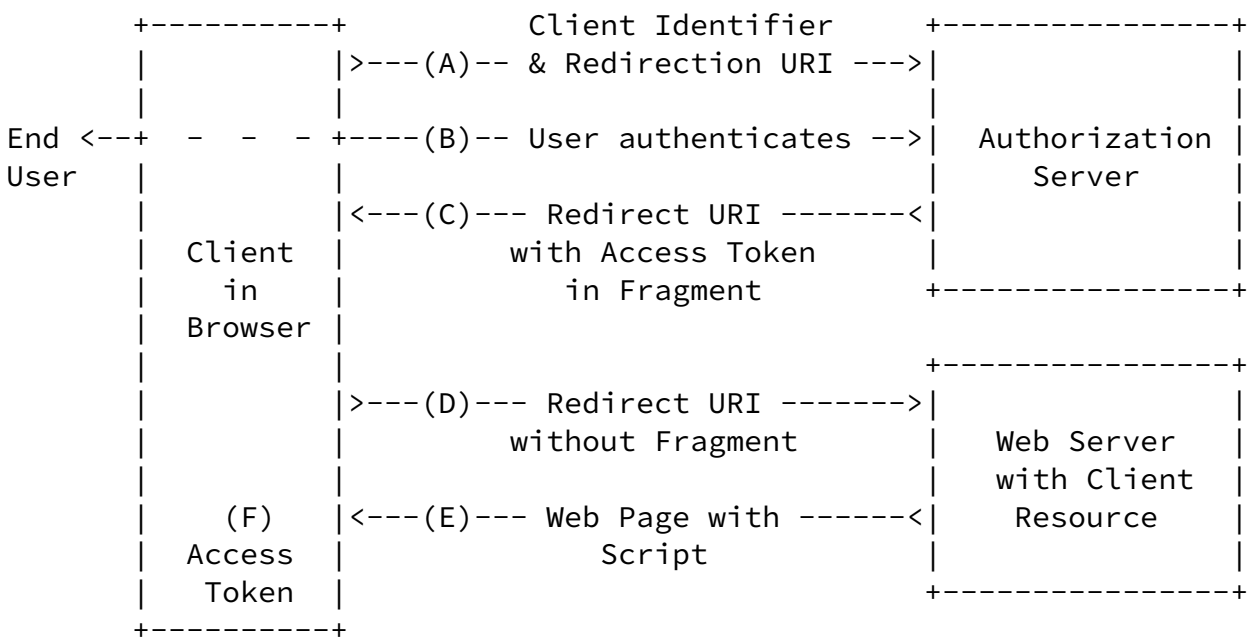


Figure 6: User-Agent Flow

The user-agent flow illustrated in Figure 6 includes the following steps:

(A) The client sends the user-agent to the end-user authorization endpoint as described in Section 4. The client includes its client identifier, requested scope, local state, and a redirect URI to which the authorization server will send the end-user back once authorization is granted (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) If 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 (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.

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

2.3. Native Application

Native applications are clients running as native code on the end-user's computer or device (i.e. executing outside a user-agent or as a desktop program). These clients are often capable of interacting with (or embedding) the end-user's user-agent but are limited in how such interaction affects their end-user experience. In many cases, native applications are incapable of receiving direct callback requests from the server (e.g. firewall, operating system restrictions).

Native application clients can be implemented in different ways based on their requirements and desired end-user experience. Native application clients can:

- Utilize the end-user authorization endpoint as described in Section 4 by launching an external user-agent. The client can capture the response by providing a redirection URI with a custom URI scheme (registered with the operating system to invoke the client application), or by providing a redirection URI pointing to a server-hosted resource under the client's control which makes the response available to the client (e.g. using the window title or other locations accessible from outside the user-agent).

- Utilize the end-user authorization endpoint as described in Section 4 by using an embedded user-agent. The client obtains the response by directly communicating with the embedded user-agent.

- Prompt end-users for their password and use them directly to obtain an access token. This is generally discouraged, as it hands the end-user's password directly to the third-party client which in turn has to store it in clear-text. It also requires the server to support password-based authentication.

When choosing between launching an external browser and an embedded user-agent, developers should consider the following:

- External user-agents may improve completion rate as the end-user may already be logged-in and not have to re-authenticate.

- Embedded user-agents often offer a better end-user flow, as they remove the need to switch context and open new windows.

- Embedded user-agents pose a security challenge because users are authenticating in an unidentified window without access to the visual protections offered by many user-agents.

2.4. Autonomous

Autonomous clients utilize an existing trust relationship or framework to establish authorization. Autonomous clients can be implemented in different ways based on their requirements and the existing trust framework they rely upon. Autonomous clients can:

- Obtain an access token by authenticating with the authorization server using their client credentials. The scope of the access token is limited to the protected resources under the control of the client, or that of another resource owner previously arranged with the authorization server.

- Use an existing access grant expressed as an assertion using an assertion format supported by the authorization server. Using

assertions requires the client to obtain an assertion (such as a

SAML [OASIS.saml-core-2.0-os] assertion) from an assertion issuer or to self-issue an assertion. 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.

3. Client Credentials

When interacting with the authorization server, the client identifies itself using a set of client credentials which include a client identifier and other properties for client authentication. The means through which the client obtains its credentials are beyond the scope of this specification, but typically involve registration with the authorization server.

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

The authorization server MAY authenticate the client using any appropriate set of credentials and authentication schemes. The client MUST NOT include more than one set of credentials or authentication mechanism with each request.

3.1. Client Password Credentials

The client password credentials use a shared symmetric secret to authenticate the client. The client identifier and password are included in the request using the HTTP Basic authentication scheme as defined in [RFC2617] by including the client identifier as the username and client password as the password.

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

```
POST /token HTTP/1.1
```

```
  Host: server.example.com
  Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
  Content-Type: application/x-www-form-urlencoded

  grant_type=authorization_code&code=i1WsRn1uB1&
  redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```


Alternatively, the client MAY include the password in the request body using the following parameters:

client_id
: REQUIRED. The client identifier.

client_secret REQUIRED. The client password.

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

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

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

The authorization server MUST accept the client credentials using both the request parameter, and the HTTP Basic authentication scheme. The authorization server MAY support additional authentication schemes suitable for the transmission of password credentials.

3.2. Client Assertion Credentials

The client assertion credentials are used in cases where a password (clear-text shared symmetric secret) is unsuitable or does not provide sufficient security for client authentication. In such cases it is common to use other mechanisms such as HMAC or digital signatures that do not require sending clear-text secrets. The client assertion credentials provide an extensible mechanism to use

an assertion format supported by the authorization server for authentication the client.

Using assertions requires the client to obtain an assertion (such as a SAML [OASIS.saml-core-2.0-os] assertion) from an assertion issuer or to self-issue an assertion. 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.

When using a client assertion, the client includes the following parameters:

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

client_assertion REQUIRED. The client assertion.

For example, the client sends the following access token request using a SAML 2.0 assertion to authenticate itself (line breaks are for display purposes only):

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

grant_type=authorization_code&code=i1WsRn1uB1&
client_assertion=PHNhbWxwOl[...omitted for brevity...]ZT4%3D&
client_assertion_type=
urn%3Aoasis%3Anames%sAtc%3ASAML%3A2.0%3Aassertion&
redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

When obtaining an access token using a client assertion together with an authorization code (as described in Section 5.1.1), a mechanism is needed to map between the value of "client_id" parameter used to obtain the authorization code, and the client assertion. Such

mechanism is beyond the out of scope for this specification, but MUST be specified for any client assertion type used in combination with an authorization code.

The authorization server MUST reject access token requests using client assertion credentials that do not contain HMAC or signed values that:

- State the assertion was specifically issued to be consumed by the receiving endpoint (typically via an audience or recipient value containing the receiving endpoint's identifier).
- Identify the entity that issued the assertion (typically via an issuer value).
- Identify when the assertion expires as an absolute time (typically via an expiration value containing a UTC date/time value). The authorization server MUST reject expired assertions.

4. Obtaining End-User Authorization

Before the client can access a protect resource, it MUST first obtain authorization from the end-user. To obtain an end-user authorization, the client sends the end-user to the end-user authorization endpoint. Once obtained, the end-user access grant is expressed as an authorization code which the client uses to obtain an access token.

At the end-user authorization endpoint, the end-user first authenticates with the authorization server, and then grants or denies the access request. 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, is beyond the scope of this specification. However, the authorization server MUST first verify the identity of the end-user.

The location of the end-user authorization endpoint can be found in

the service documentation. The end-user authorization endpoint URI MAY include a query component as defined by [RFC3986] section 3, which must be retained when adding additional query parameters.

Since requests to the end-user authorization endpoint result in user authentication and the transmission of sensitive information, the authorization server SHOULD require the use of a transport-layer security mechanism such as TLS when sending requests to the end-user authorization endpoint.

4.1. Authorization Request

In order to direct the end-user's user-agent to the authorization server, the client constructs the request URI by adding the following parameters to the end-user authorization endpoint URI query component using the "application/x-www-form-urlencoded" format as defined by [W3C.REC-html401-19991224]:

response_type
: REQUIRED. The requested response: an access token, an authorization code, or both. The parameter value MUST be set to "token" for requesting an access token, "code" for requesting an authorization code, or "code_and_token" to request both. The authorization server MAY decline to provide one or more of these response types.

client_id
: REQUIRED. The client identifier as described in Section 3.

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.

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.

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.

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 authorization server MUST support the use of the HTTP "GET" method for the end-user authorization endpoint, and MAY support the use of the "POST" method as well.

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

```
GET /authorize?response_type=code&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 SHOULD NOT redirect the user-agent to unregistered or untrusted URIs to prevent the endpoint from being used as an open redirector. If no valid redirection URI is available, the authorization server SHOULD inform

the end-user of the error occured. [[provide guidance on how to perform matching]]

Parameters sent without a value MUST be treated as if they were omitted from the request. The authorization server SHOULD ignore

unrecognized request parameters.

The authorization server validates the request to ensure all required parameters are present and valid. If the request is invalid, the authorization server redirects the user-agent back to the client using the redirection URI provided with the appropriate error code as described in Section 4.3.

The authorization server authenticates the end-user and obtains an authorization decision (by asking the end-user or by establishing approval via other means). When a decision has been established, the authorization server directs 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.

4.2. Authorization Response

If the end-user grants the access request, the authorization server issues an access token, an authorization code, or both, and delivers them to the client by adding the following parameters to the redirection URI (as described below):

code
: REQUIRED if the response type is "code" or "code_and_token", otherwise MUST NOT be included. The authorization code generated by the authorization server. The authorization code SHOULD expire shortly after it is issued to minimize the risk of leaks. The client MUST NOT reuse the authorization code. If an authorization code is used more than once, the authorization server MAY revoke all tokens previously issued based on that authorization code. The authorization code is bound to the client identifier and redirection URI.

access_token
: REQUIRED if the response type is "token" or "code_and_token", otherwise MUST NOT be included. The access token issued by the authorization server.

token_type
: REQUIRED if the response includes an access token. The type of the token issued. The token type informs the client how the access token is to be used when accessing a protected resource as described in Section 6.1.

expires_in
: OPTIONAL. The duration in seconds of the access token lifetime if an access token is included. For example, the value "3600" denotes that the access token will expire in one hour from the time the response was generated by the authorization server.

scope
: OPTIONAL. The scope of the access token as a list of space-delimited strings if an access token is included. 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. The authorization server SHOULD include the parameter if the requested scope is different from the one requested by the client.

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

The method in which the authorization server adds the parameter to the redirection URI is determined by the response type requested by the client in the authorization request using the "response_type" parameter.

If the response type is "code", the authorization server adds the parameters to the redirection URI query component using the "application/x-www-form-urlencoded" format as defined by [W3C.REC-html401-19991224].

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
```

If the response type is "token" or "code_and_token", the authorization server adds the parameters to the redirection URI fragment component using the "application/x-www-form-urlencoded" format as defined by [W3C.REC-html401-19991224].

For example, the authorization server redirects the end-user's user-agent by sending the following HTTP response (URI line breaks are for display purposes only):

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

Clients SHOULD ignore unrecognized response 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.

4.3. Error Response

If the end-user denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding the following parameters to the redirection URI query component using the "application/x-www-form-urlencoded" format as defined by [W3C.REC-html401-19991224]:

error
: REQUIRED. A single error code as described in Section 4.3.1.

error_description
: OPTIONAL. A human-readable text providing additional information, used to assist in the understanding and resolution of the error occurred.

error_uri
: OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the end-user with additional information about the error.

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 redirects the end-user's user-

agent by sending the following HTTP response:

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


If the request fails due to a missing or invalid redirection URI, the authorization server SHOULD inform the end-user of the error, and MUST NOT redirect the end-user's user-agent to the invalid redirection URI.

4.3.1. Error Codes

The authorization server includes one of the following error codes with the error response:

invalid_request
: The request is missing a required parameter, includes an unsupported parameter or parameter value, or is otherwise malformed.

invalid_client
: The client identifier provided is invalid.

unauthorized_client
: The client is not authorized to use the requested response type.

redirect_uri_mismatch
: The redirection URI provided does not match a pre-registered value.

access_denied
: The end-user or authorization server denied the request.

unsupported_response_type
: The requested response type is not supported by the authorization server.

invalid_scope
: The requested scope is invalid, unknown, or malformed.

[[Add mechanism for extending error codes]]

5. Obtaining an Access Token

The client obtains an access token by authenticating with the authorization server and presenting its access grant (in the form of an authorization code, resource owner credentials, an assertion, or a refresh token).

Since requests to the token endpoint result in the transmission of clear-text credentials in the HTTP request and response, the

authorization server MUST require the use of a transport-layer security mechanism when sending requests to the token endpoints. Servers MUST support TLS 1.2 as defined in [RFC5246], and MAY support additional transport-layer security mechanisms.

The client requests an access token by making an HTTP "POST" request to the token endpoint. The location of the token endpoint can be found in the service documentation. The token endpoint URI MAY include a query component.

The client authenticates with the authorization server by adding its client credentials to the request as described in Section 3. The authorization server MAY allow unauthenticated access token requests when the client identity does not matter (e.g. anonymous client) or when the client identity is established via other means (e.g. using an assertion access grant).

The client constructs the request by including the following parameters using the "application/x-www-form-urlencoded" format in the HTTP request entity-body:

grant_type
: REQUIRED. The access grant type included in the request. Value MUST be one of "authorization_code", "password", "refresh_token", "client_credentials", or an absolute URI identifying an assertion format supported by the authorization server.

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. If the access grant being used already represents an approved scope (e.g. authorization code, assertion), the requested scope MUST be equal or lesser than the scope previously granted, and if omitted is treated as equal to the previously approved scope.

In addition, the client MUST include the appropriate parameters listed for the selected access grant type as described in Section 5.1.

Parameters sent without a value MUST be treated as if they were omitted from the request. The authorization server SHOULD ignore unrecognized request parameters.

5.1. Access Grant Types

The client requests an access token using an authorization code, resource owner password credentials, client credentials, refresh token, or assertion.

5.1.1. Authorization Code

The client includes the authorization code using the "authorization_code" access grant type and the following parameters:

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

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

For example, the client makes the following HTTP request by including its client credentials via the "client_secret" parameter described in Section 3 and using transport-layer security (line breaks are for

display purposes only):

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

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

The authorization server MUST:

- Validate the client credentials (if present) and ensure they match the authorization code.

- Verify that the authorization code 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 5.2.

5.1.2. Resource Owner Password Credentials

The client includes the resource owner credentials using the "password" access grant type and the following parameters: [[add

internationalization consideration for username and password]]

username
: REQUIRED. The resource owner's username.

password
: REQUIRED. The resource owner's password.

For example, the client makes the following HTTP request by including its client credentials via the "client_secret" parameter described in Section 3 and using transport-layer security (line breaks are for display purposes only):

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

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

The authorization server MUST validate the client credentials (if present) and end-user credentials and if valid issue an access token response as described in Section 5.2.

5.1.3. Client Credentials

The client can request an access token using only its client credentials using the "client_credentials" access grant type. When omitting an explicit access grant, the client is requesting access to the protected resources under its control, or those of another resource owner which has been previously arranged with the authorization server (the method of which is beyond the scope of this specification).

5.1.4. Refresh Token

The client includes the refresh token using the "refresh_token" access grant type and the following parameter:

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

For example, the client makes the following HTTP request by including its client credentials via the "client_secret" parameter described in Section 3 and using transport-layer security (line breaks are for display purposes only):

```
POST /token HTTP/1.1
```

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

  grant_type=refresh_token&client_id=s6BhdRkqt3&
  client_secret=8eSEIpnqmM&refresh_token=n4E9O119d
```

The authorization server MUST verify the client credentials (if present), 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 5.2. The authorization server MAY issue a new refresh token, in which case, the client MUST discard the old refresh token and replace it with the new refresh token.

5.1.5. Assertion

The client includes an assertion by specifying the assertion format using an absolute URI (as defined by the authorization server) as the value of the "grant_type" parameter and by adding the following parameter:

assertion
: REQUIRED. The assertion.

For example, the client makes the following HTTP request using transport-layer security, and client authentication is achieved via the assertion (line breaks are for display purposes only):

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

  grant_type=urn%3Aoasis%3Anames%3Atc%3ASAML%3A2.0%3Aassertion&
  assertion=PHNhbWxwOl[...omitted for brevity...]ZT4%3D
```

The authorization server MUST validate the client credentials (if present) and the assertion and if valid issues an access token response as described in Section 5.2. The authorization server

SHOULD NOT issue a refresh token (instead, it should require the client to use the same or new 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.

5.2. Access Token Response

After receiving and verifying a valid and authorized access token request from the client, the authorization server issues the access token and optional refresh token, and constructs the response by adding the following parameters to the entity body of the HTTP response with a 200 (OK) status code:

The token response contains the following parameters:

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

token_type
: REQUIRED. The type of the token issued. The token type informs the client how the access token is to be used when accessing a protected resource as described in Section 6.1.

expires_in
: OPTIONAL. The duration in seconds of the access token lifetime. For example, the value "3600" denotes that the access token will expire in one hour from the time the response was generated by the authorization server.

refresh_token
: OPTIONAL. The refresh token used to obtain new access tokens using the same end-user access grant as described in Section 5.1.4. The authorization server SHOULD NOT issue a refresh token when the access grant type is an assertion or a set of client credentials.

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. The authorization server SHOULD include the parameter if the requested scope is different from the one requested by the client.

The parameters are including in the entity body of the HTTP response using 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 values 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.

For example:

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

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

Clients SHOULD ignore unrecognized response 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.

5.3. Error Response

If the token request is invalid or unauthorized, the authorization server constructs the response by adding the following parameter to the entity body of the HTTP response using the "application/json" media type:

error
: REQUIRED. A single error code as described in Section 5.3.1.

error_description
: OPTIONAL. A human-readable text providing

additional information, used to assist in the understanding and resolution of the error occurred.

error_uri OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the end-user with additional information about the error.

For example:

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

{
  "error":"invalid_request"
}
```

If the client provided invalid credentials using an HTTP authentication scheme via the "Authorization" request header field, the authorization server MUST respond with the HTTP 401 (Unauthorized) status code. Otherwise, the authorization server SHALL respond with the HTTP 400 (Bad Request) status code.

5.3.1. Error Codes

The authorization server includes one of the following error codes with the error response:

invalid_request
: The request is missing a required parameter, includes an unsupported parameter or parameter value, repeats a parameter, includes multiple credentials, utilizes more than one mechanism for authenticating the client, or is otherwise malformed.

invalid_client
: The client identifier provided is invalid, the client failed to authenticate, the client did not include its credentials,

provided multiple client credentials, or used unsupported credentials type.

unauthorized_client
: The authenticated client is not authorized to use the access grant type provided.

invalid_grant
: The provided access grant is invalid, expired, or revoked (e.g. invalid assertion, expired authorization token, bad end-user password credentials, or mismatching authorization code and redirection URI).

unsupported_grant_type
: The access grant included - its type or another attribute - is not supported by the authorization server.

invalid_scope
: The requested scope is invalid, unknown, malformed, or exceeds the previously granted scope.

[[Add mechanism for extending error codes]]

6. Accessing a Protected Resource

Clients access protected resources by presenting an access token to the resource server. The resource server MUST validate the access token and ensure it has not expired and that its scope covers the requested resource. 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 the client utilized the access token to authenticate with the resource server depends on the type of access token issued by the authorization server.

6.1. Access Token Types

[[add token type explanation, maybe with links to other token specs]]

6.2. The WWW-Authenticate Response Header Field

If the protected resource request does not include authentication credentials, contains an invalid access token, or is malformed, the resource server MUST include the HTTP "WWW-Authenticate" response header field. The "WWW-Authenticate" header field uses the framework defined by [RFC2617] as follows:


```
challenge       = "OAuth2" [ RWS 1#param ]

param           = scope /
                  error / error-desc / error-uri /
                  ( token "=" ( token / quoted-string ) )

scope           = "scope" "=" <"> scope-v *( SP scope-v ) <">
scope-v         = 1*quoted-char

quoted-char     = ALPHA / DIGIT /
                  "!" / "#" / "$" / "%" / "&" / "'" / "(" / ")" /
                  "*" / "+" / "-" / "." / "/" / ":" / "<" / "=" /
                  ">" / "?" / "@" / "[" / "]" / "^" / "_" / "`" /
                  "{" / "|" / "}" / "~" / "\" / "," / ";"

error           = "error" "=" quoted-string
error-desc      = "error_description" "=" quoted-string
error-uri       = "error_uri" = <"> URI-reference <">
```

The "scope" attribute is a space-delimited list of scope values indicating the required scope of the access token for accessing the

requested resource. The "scope" attribute MUST NOT appear more than once.

If the protected resource request included an access token and failed authentication, the resource server SHOULD include the "error" attribute to provide the client with the reason why the access request was declined. The parameter value is described in Section 6.2.1. In addition, the resource server MAY include the "error_description" attribute to provide a human-readable explanation, and the "error-uri" attribute with an absolute URI identifying a human-readable web page explaining the error. The "error", "error_description", and "error_uri" attribute MUST NOT appear more than once.

For example, in response to a protected resource request without authentication:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: OAuth2
```


And in response to a protected resource request with an authentication attempt using an expired access token:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: OAuth2
                  error="invalid_token",
                  error_description="The access token expired"
```

6.2.1. Error Codes

When a request fails, the resource server responds using the appropriate HTTP status code (typically, 400, 401, or 403), and includes one of the following error codes in the response:

invalid_request
: The request is missing a required parameter, includes an unsupported parameter or parameter value, repeats the same parameter, uses more than one method for including an access token, or is otherwise malformed. The resource server SHOULD respond with the HTTP 400 (Bad Request) status code.

invalid_token
: The access token provided is expired, revoked, malformed, or invalid for other reasons. The resource SHOULD respond with the HTTP 401 (Unauthorized) status code. The client MAY request a new access token and retry the protected resource request.

insufficient_scope
: The request requires higher privileges than provided by the access token. The resource server SHOULD respond with the HTTP 403 (Forbidden) status code and MAY include the "scope" attribute with the scope necessary to access the protected resource.

[[Add mechanism for extending error codes]]

If the request lacks any authentication information (i.e. the client was unaware authentication is necessary or attempted using an unsupported authentication method), the resource server SHOULD not include an error code or other error information.

For example:

```
  HTTP/1.1 401 Unauthorized
```



```
  WWW-Authenticate: OAuth2
```

7. Extensibility

7.1. Defining New Client Credentials Types

[[TBD]]

7.2. Defining New Endpoint Parameters

Applications that wish to define new request or response parameters for use with the end-user authorization endpoint or the token endpoint SHALL do so in one of two ways: register them in the parameters registry (following the procedures in Section 9.1), or use the "x_" parameter name prefix.

Parameters utilizing the "x_" parameter name prefix MUST be limited to vendor-specific extensions that are not commonly applicable, and are specific to the implementation details of the authorization server where they are used. All other new parameters MUST be registered, and MUST NOT use the "x_" parameter name prefix.

Parameter names MUST conform to the param-name ABNF, and parameter values syntax MUST be well-defined (e.g., using ABNF, or a reference to the syntax of an existing parameter).

```
param-name  = 1*name-char
name-char   = "-" / "." / "_" / DIGIT / ALPHA
```

7.3. Defining New Header Field Parameters

Applications that wish to define new parameters for use in the OAuth "WWW-Authenticate" header field MUST register them in the parameters registry, following the procedures in Section 9.1.

Parameter names MUST conform to the param-name ABNF and MUST NOT begin with "x_". Parameter values MUST conform to the param-value ABNF and their syntax MUST be well-defined (e.g., using ABNF, or a reference to the syntax of an existing parameter).

```
param-value  = quoted-value | quoted-string
```


7.4. Defining New Access Grant Types

The assertion access grant type allows the authorization server to accept additional access grants not specified. Applications that wish to define additional access grant types can do so by utilizing a new or existing assertion type and format.

8. Security Considerations

[[TBD]]

9. IANA Considerations

9.1. The OAuth Parameters Registry

This document establishes the OAuth parameters registry.

Additional parameters to be use in the end-user authorization endpoint request, the end-user authorization endpoint response, the token endpoint request, the token endpoint response, or the "WWW-Authenticate" header field, are registered on the advice of one or more Designated Experts (appointed by the IESG or their delegate), with a Specification Required (using terminology from [RFC5226]). However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests should be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for parameter: example"). [[Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: oauth-ext-review.]]

Before a period of 14 days has passed, the Designated Expert(s) will either approve or deny the registration request, communicating this decision both to the review list and to IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful. Registration requests that are undetermined for a period longer than 21 days can be brought to the IESG's attention (using the iesg@iesg.org mailing list) for resolution.

9.1.1. Registration Template

Parameter name: The name requested (e.g., "example").

Parameter usage location: The location(s) where parameter can be used. The possible locations are: the end-user authorization endpoint request, the end-user authorization endpoint response, the token endpoint request, the token endpoint response, the or the "WWW-Authenticate" header field.

Change controller: For standards-track RFCs, state "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, e-mail address, home page URI) may also be included.

Specification document(s): Reference to document that specifies the parameter, preferably including a URI that can be used to retrieve a copy of the document. An indication of the relevant sections may also be included, but is not required.

Related information: Optionally, citations to additional documents containing further relevant information.

9.1.2. Example

The following is the parameter registration request for the "scope" parameter as defined in this specification:

Parameter name: scope

Parameter usage location: The end-user authorization endpoint request, the end-user authorization endpoint response, the token endpoint request, the token endpoint response, and the "WWW-Authenticate" header field.

Change controller: IETF

Specification document(s): [[this document]]

Related information: None

Appendix A. Examples

[[TBD]]

Appendix B. Contributors

The following people contributed to preliminary versions of this

document: Blaine Cook (BT), Brian Eaton (Google), Yaron Goland (Microsoft), Brent Goldman (Facebook), Raffi Krikorian (Twitter), Luke Shepard (Facebook), and Allen Tom (Yahoo!). The content and concepts within are a product of the OAuth community, WRAP community, and the OAuth Working Group.

The OAuth Working Group has dozens of very active contributors who proposed ideas and wording for this document, including: [[If your name is missing or you think someone should be added here, please send Eran a note - don't be shy]]

Michael Adams, Andrew Arnott, Dirk Balfanz, Brian Campbell, Leah Culver, Bill de hOra, Brian Ellin, Igor Faynberg, George Fletcher, Tim Freeman, Evan Gilbert, Kristoffer Gronowski, Justin Hart, Mike Jones, John Kemp, Chasen Le Hara, Torsten Lodderstedt, Alastair Mair, Eve Maler, James Manger, Laurence Miao, Chuck Mortimore, Justin Richer, Peter Saint-Andre, Nat Sakimura, Rob Sayre, Marius Scurtescu, Naitik Shah, Justin Smith, Jeremy Suriel, Christian Stuebner, Paul Tarjan, Franklin Tse, and Nick Walker.

Appendix C. Acknowledgements

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

Appendix D. Document History

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

-11

- Many editorial changes. Fixed user authorization section structure. Removed unused normative references. Adjusted language regarding single use of authorization codes.

- Fixed header ABNF.

- Change access token description from shared symmetric secret to

password.

- Moved access grant 'none' to a separate section, renamed to 'client_credentials'.

- Demoted the HTTP status code requirement from MUST to SHOULD in protected resource response error.

- Removed 'expired_token' error code.

- Moved all the 'code_and_token' parameter to the fragment (from code being in the query).

- Removed 'assertion_type' parameter (moved to 'grant_type').

- Added note about redirecting to invalid redirection URIs (open redirectors).

- Removed bearer token section, added new required 'token_type' parameter with extensibility.

- 'error-uri' parameter value changed to absolute URI.

- OAuth 2.0 HTTP authentication scheme name changed to 'OAuth2'.

- Dropped the 'WWW-Authenticate' header field 'realm' parameter.

- Removed definition of access token characters.

- Added instructions for dealing with error and an invalid redirection URI.

-10

- Fixed typos. Many editorial changes. Rewrote introduction. removed terminology grouping.

- Allowed POST for end-user authorization endpoint.

- Fixed token endpoint to not require client authentication.

- Made URI query and POST body 'oauth_token' parameter optional.
- Moved all parameter names and values to use underscores.
- Changed 'basic_credentials' to 'password', 'invalid_client_credentials' and 'invalid_client_id' to 'invalid_client'.
- Added note that access token requests without an access grant should not include a refresh token.
- Changed scheme name from 'Token' to 'OAuth', simplified request format to simple string for token instead of key=value pair (still supported for extensions).

- Defined permitted access token string characters (suitable for inclusion in an HTTP header).
- Added a note about conflicts with previous versions.
- Moved 'client_id' definition from client authentication to access token endpoint.
- Added definition for 'access grant'.

-09

- Fixed typos, editorial changes.
- Added token expiration example.
- Added scope parameter to end-user authorization endpoint response.
- Added note about parameters with empty values (same as omitted).
- Changed parameter values to use '-' instead of '_'. Parameter names still use '_'.
- Changed authorization endpoint client type to response type with values: code, token, and both.

- Complete cleanup of error codes. Added support for error description and URI.

- Add initial extensibility support.

-08

- Renamed verification code to authorization code.

- Revised terminology, structured section, added new terms.

- Changed flows to profiles and moved to introduction.

- Added support for access token rescoping.

- Cleaned up client credentials section.

- New introduction overview.

- Added error code for invalid username and password, and renamed error code to be more consistent.

- Added access grant type parameter to token endpoint.

-07

- Major rewrite of entire document structure.

- Removed device profile.

- Added verification code support to user-agent flow.

- Removed multiple formats support, leaving JSON as the only format.

- Changed assertion "assertion_format" parameter to "assertion_type".

- Removed "type" parameter from token endpoint.

-06

- Editorial changes, corrections, clarifications, etc.
- Removed conformance section.
- Moved authors section to contributors appendix.
- Added section on native applications.
- Changed error response to use the requested format. Added support for HTTP "Accept" header.
- Flipped the order of the web server and user-agent flows.
- Renamed assertion flow "format" parameter name to "assertion_format" to resolve conflict.
- Removed the term identifier from token definitions. Added a cryptographic token definition.
- Added figure titles.
- Added server response 401 when client tried to authenticate using multiple credentials.
- Clarified support for TLS alternatives, and added requirement for TLS 1.2 support for token endpoint.
- Removed all signature and cryptography.

- Removed all discovery.
- Updated HTML4 reference.

-05

- Corrected device example.
- Added client credentials parameters to the assertion flow as OPTIONAL.

- Added the ability to send client credentials using an HTTP authentication scheme.
- Initial text for the "WWW-Authenticate" header (also added scope support).
- Change authorization endpoint to end-user endpoint.
- In the device flow, change the "user_uri" parameter to "verification_uri" to avoid confusion with the end-user endpoint.
- Add "format" request parameter and support for XML and form-encoded responses.

-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.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[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.

[RFC2828] Shirey, R., "Internet Security Glossary", RFC 2828, May 2000.

[RFC3023] Murata, M., St. Laurent, S., and D. Kohn, "XML Media Types", RFC 3023, January 2001.

[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.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5849] Hammer-Lahav, E., "The OAuth 1.0 Protocol", RFC 5849, April 2010.

[W3C.REC-html401-19991224]
Raggett, D., Hors, A., and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, December 1999, <http://www.w3.org/TR/1999/REC-html401-19991224>.

10.2. Informative References

[OASIS.saml-core-2.0-os]
Cantor, S., Kemp, J., Philpott, R., and E. Maler, "Assertions and Protocol for the OASIS Security Assertion

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
Microsoft

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