

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: December 17, 2010

E. Hammer-Lahav, Ed.
Yahoo!
D. Recordon
Facebook
D. Hardt
Microsoft
June 15, 2010

The OAuth 2.0 Protocol
draft-ietf-oauth-v2-08

Abstract

This specification describes the OAuth 2.0 protocol.

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 December 17, 2010.

Copyright Notice

Copyright (c) 2010 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	3
1.1.	Notational Conventions	3
1.2.	Terminology	4
1.3.	Overview	5
1.4.	Client Profiles	8
1.4.1.	Web Server	8
1.4.2.	User-Agent	9
1.4.3.	Native Application	11
1.4.4.	Autonomous	12
2.	Client Credentials	12
2.1.	Basic Client Credentials	13
3.	Obtaining End-User Authorization	14
3.1.	Authorization Server Response	16
4.	Obtaining an Access Token	17
4.1.	Access Grant Parameters	18
4.1.1.	Authorization Code	18
4.1.2.	Resource Owner Basic Credentials	19
4.1.3.	Assertion	20
4.1.4.	Refresh Token	20
4.2.	Access Token Response	21
4.3.	Error Response	22
4.3.1.	Error Codes	23
5.	Accessing a Protected Resource	23
5.1.	The Authorization Request Header Field	24
5.2.	URI Query Parameter	25
5.3.	Form-Encoded Body Parameter	25
6.	The WWW-Authenticate Response Header Field	26
7.	Security Considerations	27
8.	IANA Considerations	27
Appendix A.	Examples	27
Appendix B.	Contributors	27
Appendix C.	Acknowledgements	28
Appendix D.	Document History	28
9.	References	31
9.1.	Normative References	31
9.2.	Informative References	32
	Authors' Addresses	32

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, a client accessing a protected resource on a server presents the resource owner's credentials in order to authenticate and gain access.

OAuth introduces a third role to the traditional client-server authentication model: the resource owner. In OAuth, the client (which is usually not the resource owner, but is acting on its behalf) requests access to resources controlled by the resource owner and hosted by the resource server.

In addition to removing the need for resource owners to share their credentials, resource owners 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 methods supported by these resources.

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

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 as defined by [[RFC2818](#)]). Other specifications may extend it for use with other transport protocols.

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 realm and auth-param rules are included from [[RFC2617](#)].

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 and can self-contain the authorization information in a verifiable manner (i.e. signed), or denotes an identifier used to retrieve the information. Tokens represent a specific scope, duration, and other authorization attributes granted by the resource owner and enforced by the resource server and authorization servers.

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 (in addition or as a replacement for an expired access token), without having to involve the resource owner.

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

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.

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 3](#).

token endpoint

The authorization server's HTTP endpoint capable of issuing tokens and refreshing expired tokens. The token endpoint is described in [Section 4](#).

client identifier

An 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 2](#).

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 from the resource owner, then exchange that 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.

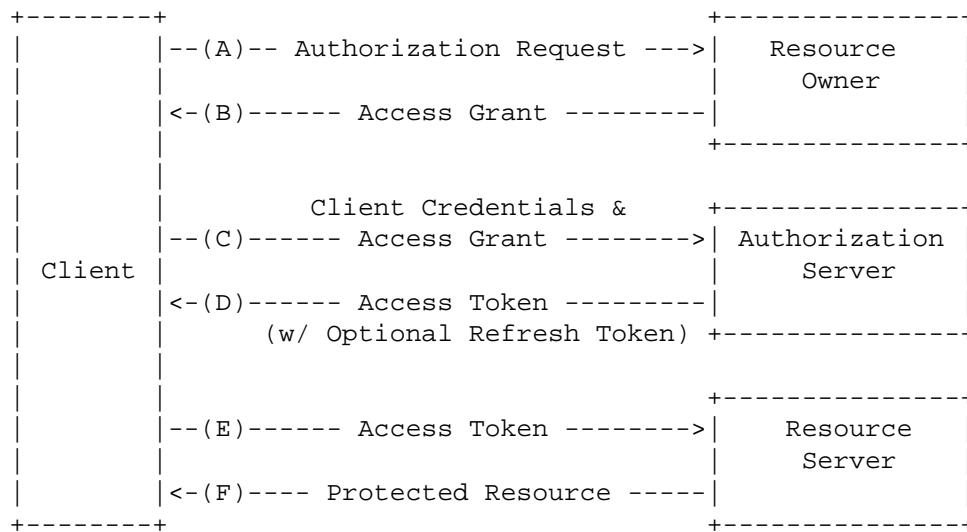


Figure 1: Abstract Protocol Flow

The abstract flow illustrated in Figure 1 includes the following steps:

- (A) The client requests authorization from the resource owner. The client should not interact directly with the resource owner (since that would exposing the resource owner's credentials to the client), but instead requests authorization via an authorization server or other entities. For example, the client directs the resource owner to the authorization server which in turn issues it an access grant. When cannot be avoided, the client interacts directly with the end-user, asking for the end-user's username and password.
- (B) The client is issued an access grant which represents the authorization provided by the resource owner. The access grant can be expressed as:
 - * Authorization code - an access grant obtained via an authorization server. The process used to obtain an authorization code is described in [Section 3](#).
 - * Assertion - an access grant obtained from entities using a different trust framework. Assertions enable the client to utilize existing trust relationships to obtain an access token. They provide a bridge between OAuth and other trust frameworks. 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.

- * Basic end-user credentials - obtained when interacting directly with an end-user. Resource owner credentials should only be used when there is a high degree of trust between the resource owner the client (e.g. its computer operating system or a highly privileged application). 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.
- (C) The client requests an access token by authenticating with the authorization server, and presenting the access grant. The token request is described in [Section 4](#).
- (D) The authorization server validated the client credentials and the access grant, and issues an access token with an optional refresh token. Access token usually have a shorter lifetime than the access grant. Refresh tokens usually have a lifetime equal to the duration of the access grant. When an access token expires, the refresh token is used to obtain a new access token without having to request another access grant from the resource owner (in which case, the refresh token acts as an access grant).
- (E) The client makes a protect resource request to the resource server, and presents the access token in order to gain access. Accessing a protected resource is described in [Section 5](#).
- (F) The resource server validates the access token, and if valid, serves the request.

When the client is acting on behalf of itself (the client is also the resource owner), the client skips steps (A) and (B), and does not include an access grant in step (C). When the client uses the user-agent profile (described in [Section 1.4.2](#)), the authorization request (A) results in an access token (D), skipping steps (B) and (C).

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.

1.4. Client Profiles

OAuth supports a wide range of client types by providing a rich and extensible framework for establishing authorization and exchanging it for an access token. The methods detailed in this specification were designed to accomodate four client types: web servers, user-agents, native applications, and autonomous clients. Additional authorization flows and client profiles may be defined by other specifications to cover additional scenarios and client types.

1.4.1. 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 from the authorization server (capable of acting as an HTTP server).

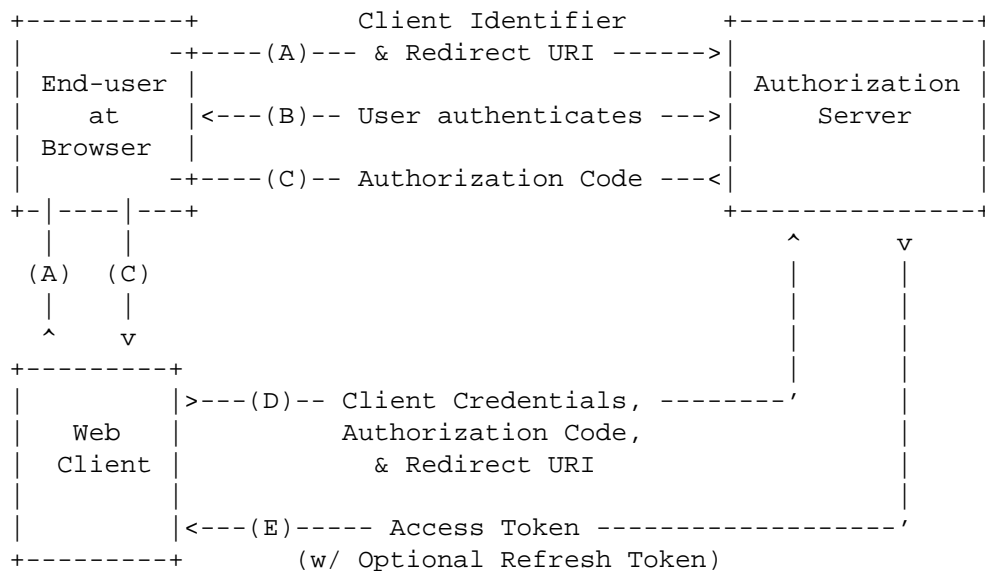


Figure 2: Web Server Flow

The web server flow illustrated in Figure 2 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 3](#) using client type "web_server". 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 4](#).
- (E) The authorization server validates the client credentials and the authorization code and responds back with the access token.

1.4.2. 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 a separate end-user authorization request and an access token requests, 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.

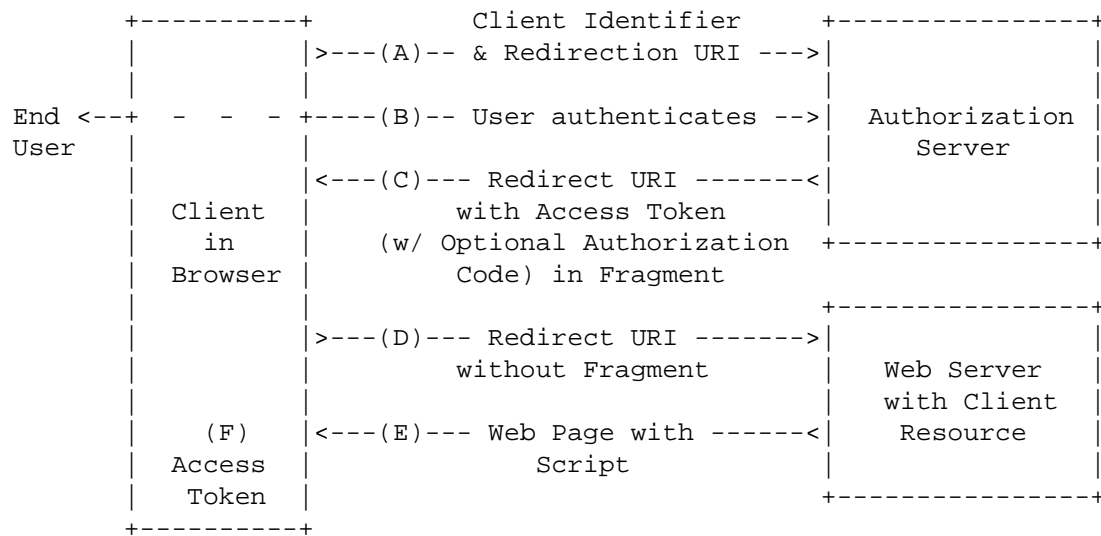


Figure 3: User-Agent Flow

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

- (A) The client sends the user-agent to the end-user authorization endpoint as described in [Section 3](#) using client type "user-agent". 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 (and an optional authorization code) 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 which extracts the access token and passes it to the client. If an authorization code was issued, the client can pass it to a web server component to obtain another access token for additional server-based protected resources interaction.

1.4.3. Native Application

Native application 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 incapable of receiving callback requests from the server (incapable of acting as an HTTP server).

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

- o Utilize the end-user authorization endpoint as described in [Section 3](#) 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 puts the response in the user-agent window title (from which the client can obtain the response by polling the user-agent window, looking for a window title change).
- o Utilize the end-user authorization endpoint as described in [Section 3](#) by using an embedded user-agent. The client obtains the response by directly communicating with the embedded user-agent.
- o Prompt end-users for their basic credentials (username and 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 3rd party and is limited to basic credentials.

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

- o External user-agents may improve completion rate as the end-user may already be logged-in and not have to re-authenticate.
- o Embedded user-agents often offer a better end-user flow, as they remove the need to switch context and open new windows.

- o Embedded user-agents are less secure because users are authenticating in unidentified window without access to the protections offered by many user-agents.

1.4.4. Autonomous

Autonomous clients act on their own behalf (the client is also the resource owner), or utilize existing trust relationship or framework to establish authorization without directly involving the resource owner.

Autonomous clients can be implemented in different ways based on their requirements and the existing trust framework they rely upon. Autonomous clients can:

- o 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.
- o 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 a 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.

2. Client Credentials

When interacting with the authorization server, the client identifies itself using a set of client credentials. The client credentials include a client identifier and MAY include a secret or other means for the authorization server to authenticate the client.

The means through which the client obtains its credentials are beyond the scope of this specification, but usually involve registration with the authorization server. [[OAuth Discovery provides one way of obtaining basic client credentials]]

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.

This specification provides one mean of authenticating the client using a set of basic client credentials. The authorization server MAY authenticate the client using any desired authentication scheme.

2.1. Basic Client Credentials

The basic client credentials include a client identifier and an OPTIONAL matching shared symmetric secret. The client identifier and secret are included in the request using the HTTP Basic authentication scheme as defined in [RFC2617] by including the client identifier as the username and secret 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

type=web_server&code=i1WsRnluB1&
redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

Alternatively, the client MAY include the credentials using the following request parameters:

```
client_id
    REQUIRED. The client identifier.

client_secret  REQUIRED if the client identifier has a matching
    secret.
```

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

type=web_server&client_id=s6BhdRkqt3&
client_secret=gXlfBat3bV&code=i1WsRnluB1&
redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

The client MAY include the client credentials using other HTTP authentication schemes which support authenticating using a username and password. The client MUST NOT include the client credentials

using more than one mechanism. If more than one mechanism is used, regardless whether the credentials are identical or valid, the server MUST reply with an HTTP 400 status code (Bad Request) and include the "multiple_credentials" error code.

The authorization server MUST accept the client credentials using both the request parameters, and the HTTP Basic authentication scheme. The authorization server MAY support additional authentication schemes.

3. Obtaining End-User Authorization

When the client interacts with an end-user, the end-user MUST first grant the client authorization to access its protected resources. Once obtained, the end-user access grant is expressed as an authorization code which the client uses to obtain an access token. To obtain an end-user authorization, the client sends the end-user to the end-user authorization endpoint.

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, or can be obtained by using [[OAuth Discovery]]. 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 mechanism such as TLS when sending requests to the end-user authorization endpoint.

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 [\[W3C.REC-html401-19991224\]](#):

type

REQUIRED. The client type (user-agent or web server). Determines how the authorization server delivers the authorization response back to the client. The parameter value MUST be set to "web_server" or "user_agent".

client_id

REQUIRED. The client identifier as described in [Section 2](#).

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.

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=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. [[provide guidance on how to perform matching]]

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.

3.1. Authorization Server 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:

code

REQUIRED if the client type is "web_server", otherwise OPTIONAL. The authorization code generated by the authorization server. The authorization code SHOULD expire shortly after it is issued and allowed for a single use. The authorization code is bound to the client identifier and redirection URI.

access_token

REQUIRED if the client type is "user_agent", otherwise MUST NOT be included. The access token.

expires_in

OPTIONAL. The duration in seconds of the access token lifetime if an access token is included.

state

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

If the end-user denies the access request, the authorization server informs the client by adding the following parameters to the redirection URI:

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.

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

If the client type is "web_server", 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=i1WsRnluB1
```

If the client type is "user_agent", 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]. [[replace form-encoded with JSON?]]

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
```

4. Obtaining an Access Token

The client obtains an access token by authenticating with the authorization server and presenting its access grant.

After obtaining authorization from the resource owner, clients request an access token from the authorization server's token endpoint. When requesting an access token, the client authenticates with the authorization server and includes the access grant (in the form of an authorization code, resource owner credentials, an assertion, or a refresh token).

The location of the token endpoint can be found in the service documentation, or can be obtained by using [\[\[OAuth Discovery \]\]](#). The token endpoint URI MAY include a query component, which must be retained when adding additional query parameters.

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 when sending requests to the token endpoints. Servers MUST support TLS 1.2 as defined in [\[RFC5246\]](#) and MAY support additional mechanisms with equivalent protections.

The client requests an access token by constructing a token request and making an HTTP "POST" request. The client constructs the request URI by adding its client credentials to the request as described in [Section 2](#), and includes 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", "user_basic_credentials", "assertion", "refresh_token", or "none" (which indicates the client is acting on behalf of itself).

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.

In addition, the client MUST include the appropriate parameters listed for the selected access grant type as described in [Section 4.1](#).

4.1. Access Grant Parameters

4.1.1. 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 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

grant_type=authorization_code&client_id=s6BhdRkqt3&
client_secret=gXlfBat3bV&code=i1WsRnluB1&
redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

The authorization server MUST verify that the authorization 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 4.2](#).

4.1.2. Resource Owner Basic Credentials

The client includes the resource owner credentials using the following parameters: [[add internationalization consideration for username and password]]

username

REQUIRED. The end-user's username.

password

REQUIRED. The end-user's password.

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

grant_type=user_basic&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 4.2](#).

4.1.3. Assertion

The client includes the assertion using the following parameters:

assertion_type

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

assertion

REQUIRED. The assertion.

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

grant_type=assertion&client_id=s6BhdRkqt3&client_secret=diejdsks&
assertion_type=urn%3Aoasis%3Anames%3Atc%3ASAML%3A2.0%3Aassertion&
assertion=PHNhbWxwOl...[omitted for brevity]...ZT4%3D
```

The authorization server MUST validate the assertion and if valid issues an access token response as described in [Section 4.2](#). The authorization server SHOULD NOT issue a refresh token.

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.

4.1.4. Refresh Token

The client includes the refresh token using the following parameters:

refresh_token

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

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

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

The authorization server MUST verify the client credentials, 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 4.2](#). 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.

4.2. 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 status code (OK):

The token response contains the following 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 4.1.4](#).

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 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",
  "expires_in": 3600,
  "refresh_token": "8xLOxBtZp8"
}
```

4.3. 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 with a a 400 status code (Bad Request) using the "application/json" media type:

```
error
    REQUIRED. The error code as described in Section 4.3.1.
```

For example:

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

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

4.3.1. Error Codes

[[explain each error code:]]

- o "redirect_uri_mismatch"
- o "bad_authorization_code"
- o "invalid_client_credentials"
- o "unauthorized_client'" - The client is not permitted to use this access grant type.
- o "invalid_assertion"
- o "unknown_format"
- o "authorization_expired"
- o "multiple_credentials"
- o "invalid_user_credentials"

5. Accessing a Protected Resource

Clients access protected resources by presenting an access token to the resource server.

For example:

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

Access tokens act as bearer tokens, where the token string acts as a shared symmetric secret. This requires treating the access token with the same care as other secrets (e.g. end-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 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.

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 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 field as described in [Section 6](#).

For example:

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

Clients make authenticated token requests using the "Authorization" request header field as described in [Section 5.1](#). Alternatively, clients MAY include the access token using the HTTP request URI in the query component as described in [Section 5.2](#), or in the HTTP body when using the "application/x-www-form-urlencoded" content type as described in [Section 5.3](#).

Clients SHOULD only use the request URI or body when the "Authorization" request header field is not available, and MUST NOT use more than one method in each request. [[specify error]]

5.1. The Authorization Request Header Field

The "Authorization" request header field is used by clients to make authenticated token requests. The client uses the "token" attribute to include the access token in the request.

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

```
credentials    = "Token" RWS access-token [ CS 1#auth-param ]
access-token   = "token" "=" <"> token <">
CS             = OWS " ," OWS
```


5.2. 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 field as described in [Section 6](#).

5.3. 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:

- o The entity-body is single-part.
- o The entity-body follows the encoding requirements of the "application/x-www-form-urlencoded" content-type as defined by [\[W3C.REC-html401-19991224\]](#).
- o The HTTP request entity-header includes the "Content-Type" header field set to "application/x-www-form-urlencoded".
- o 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 field as described in [Section 6](#).

6. The WWW-Authenticate Response Header Field

Clients access protected resources after locating the appropriate end-user authorization endpoint and token endpoint 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 end-user 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.

A resource server receiving a request for a protected resource without a valid access token MUST respond with a 401 (Unauthorized) or 403 (Forbidden) HTTP status code, and include 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 error ]
                  [ CS 1#auth-param ]

error          = "error" "=" "<"> token "<">
```

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

The "error" attribute is used to inform the client the reason why an access request was declined. [[Add list of error codes]]

7. Security Considerations

[[todo]]

8. IANA Considerations

[[Not Yet]]

Appendix A. Examples

[[todo]]

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, Igor Faynberg, George Fletcher, Evan Gilbert, Justin Hart, John Kemp, Torsten Lodderstedt, Eve Maler, James Manger, Chuck

Mortimore, Justin Richer, Peter Saint-Andre, Nat Sakimura, Rob Sayre, Marius Scurtescu, Justin Smith, and Franklin Tse.

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]]

-08

- o Renamed verification code to authorization code.
- o Revised terminology, structured section, added new terms.
- o Changed flows to profiles and moved to introduction.
- o Added support for access token rescoping.
- o Cleaned up client credentials section.
- o New introduction overview.
- o Added error code for invalid username and password, and renamed error code to be more consistent.
- o Added access grant type parameter to token endpoint.

-07

- o Major rewrite of entire document structure.
- o Removed device profile.
- o Added verification code support to user-agent flow.
- o Removed multiple formats support, leaving JSON as the only format.
- o Changed assertion "assertion_format" parameter to "assertion_type".
- o Removed "type" parameter from token endpoint.

-06

- o Editorial changes, corrections, clarifications, etc.
- o Removed conformance section.
- o Moved authors section to contributors appendix.
- o Added section on native applications.
- o Changed error response to use the requested format. Added support for HTTP "Accept" header.
- o Flipped the order of the web server and user-agent flows.
- o Renamed assertion flow "format" parameter name to "assertion_format" to resolve conflict.
- o Removed the term identifier from token definitions. Added a cryptographic token definition.
- o Added figure titles.
- o Added server response 401 when client tried to authenticate using multiple credentials.
- o Clarified support for TLS alternatives, and added requirement for TLS 1.2 support for token endpoint.
- o Removed all signature and cryptography.
- o Removed all discovery.
- o Updated HTML4 reference.

-05

- o Corrected device example.
- o Added client credentials parameters to the assertion flow as OPTIONAL.
- o Added the ability to send client credentials using an HTTP authentication scheme.
- o Initial text for the "WWW-Authenticate" header (also added scope support).
- o Change authorization endpoint to end-user endpoint.

- o In the device flow, change the "user_uri" parameter to "verification_uri" to avoid confusion with the end-user endpoint.
- o Add "format" request parameter and support for XML and form-encoded responses.

-04

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

-03

- o Fixed typo in JSON error examples.
- o Fixed general typos.
- o Moved all flows sections up one level.

-02

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

-01

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

-00

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

9. References

9.1. Normative References

- [I-D.ietf-httpbis-pl-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-pl-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.
- [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.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", [RFC 5246](#), August 2008.
- [W3C.REC-html401-19991224]
Hors, A., Raggett, D., 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>>.

9.2. 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
Microsoft

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