Network Working Group	E. Hammer-Lahav, Ed.
Internet-Draft	Yahoo!
Intended status: Standards Track	D. Recordon
Expires: December 13, 2010	Facebook
	D. Hardt
	Microsoft
	June 11, 2010

TOC

The OAuth 2.0 Protocol
draft-ietf-oauth-v2-07

Abstract

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

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.
Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.
Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as “work in progress.”
This Internet-Draft will expire on December 13, 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
    1.1. Terminology
    1.2. Overview
    1.3. Example
    1.4. Notational Conventions
2. Client Flows
    2.1. Web Server Flow
    2.2. User-Agent Flow
    2.3. Username and Password Flow
    2.4. Client Credentials Flow
    2.5. Assertion Flow
    2.6. Native Application Considerations
3. Client Credentials
    3.1. Client Authentication
4. Establishing Resource Owner Authorization
    4.1. Verification Code
        4.1.1. End-User Authorization Endpoint
    4.2. Resource Owner Credentials
    4.3. Assertion
5. Obtaining an Access Token
    5.1. Token Endpoint
        5.1.1. Verification Code
        5.1.2. Resource Owner Credentials
        5.1.3. Assertion
        5.1.4. Refresh Token
        5.1.5. Access Token Response
        5.1.6. Error Response
6. Accessing a Protected Resource
    6.1. The Authorization Request Header
    6.2. URI Query Parameter
    6.3. Form-Encoded Body Parameter
7. Identifying a Protected Resource
    7.1. The WWW-Authenticate Response Header
8. Security Considerations
9. IANA Considerations
Appendix A. Contributors
Appendix B. Acknowledgements
Appendix C. Document History
10. References
    10.1. Normative References

10.2. Informative References
§ Authors' Addresses

---

1. Introduction

TOC

With the increasing use of distributed web services and cloud computing, third-party applications require access to server-hosted resources. These resources are usually protected and require authentication using the resource owner's credentials (typically a username and password). In the traditional client-server authentication model, a client accessing a protected resource on a server presents the resource owner's credentials in order to authenticate and gain access. Resource owners should not be required to share their credentials when granting third-party applications access to their protected resources. They should also have the ability to restrict access to a limited subset of the resources they control, to limit access duration, or to limit access to the HTTP methods supported by these resources.
OAuth provides a method for making authenticated HTTP requests using a token - an identifier used to denote an access grant with specific scope, duration, and other attributes. Tokens are issued to third-party clients by an authorization server with the approval of the resource owner. Instead of sharing their credentials with the client, resource owners grant access by authenticating directly with the authorization server which in turn issues a token to the client. The client uses the token to authenticate with the resource server and gain access.
For example, a web user (resource owner) can grant a printing service (client) access to her protected photos stored at a photo sharing service (resource server), without sharing her username and password with the printing service. Instead, she authenticates directly with the photo sharing service (authorization server) which issues the printing service delegation-specific credentials (token).
This specification defines the use of OAuth over HTTP (Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1,” June 1999.) [RFC2616] (or HTTP over TLS as defined by [RFC2818] (Rescorla, E., “HTTP Over TLS,” May 2000.)). Other specifications may extend it for use with other transport protocols.

1.1. Terminology

TOC

> **resource server** An HTTP (Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1,” June 1999.) [RFC2616]

server capable of accepting authenticated resource requests using the OAuth protocol.

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

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

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

end-user A human resource owner.

token A string representing an access grant 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 authorization information.

access token A token used by the client to make authenticated requests on behalf of the resource owner. Access tokens represent a specific scope, duration, and other access attributes granted by the resource owner and enforced by the resource and authorization servers.

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

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

end-user authorization endpoint The authorization server's HTTP endpoint capable of authenticating the end-user and obtaining authorization.

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

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

1.2. Overview

TOC

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

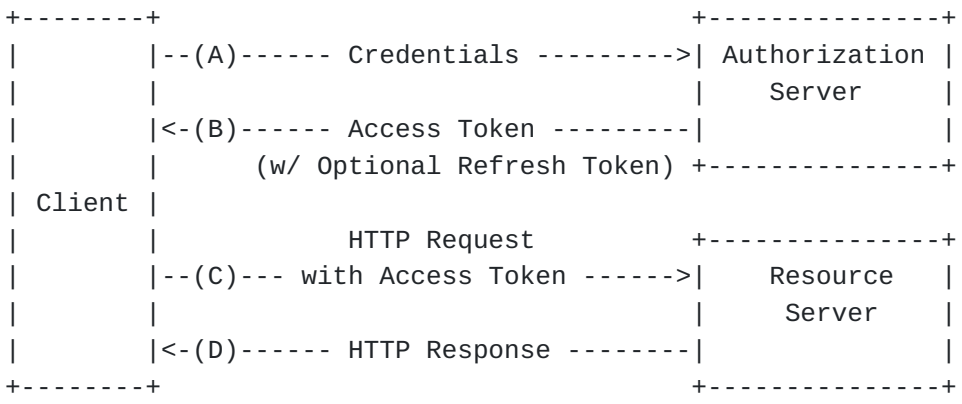

Figure 1: Generic Client-Server Flow

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

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

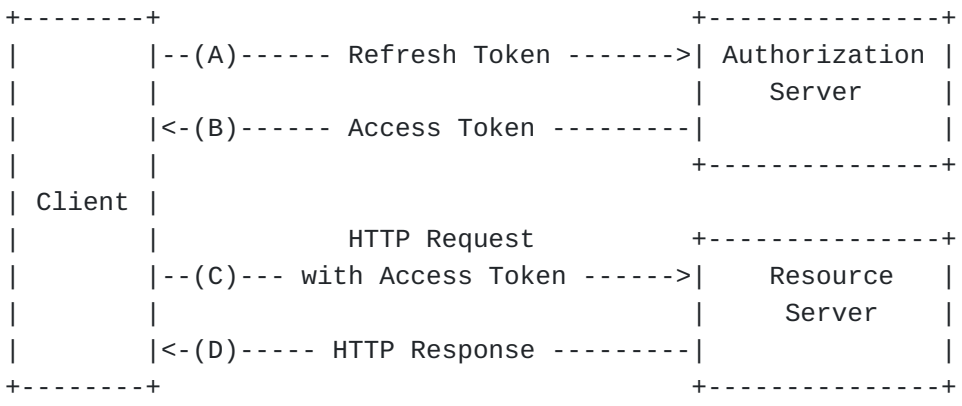

Figure 2: Refreshing an Access Token

This specification defines a number of authorization flows to support different client types and scenarios. These authorization flows can be separated into three groups: user delegation flows, direct credentials flows, and autonomous flows.
Additional authorization flows may be defined by other specifications to cover different scenarios and client types.
User delegation flows are used to grant client access to protected resources by the end-user without sharing the end-user credentials (e.g. a username and password) with the client. Instead, the end-user authenticates directly with the authorization server, and grants client access to its protected resources. The user delegation flows defined by this specifications are:

*Web Server Flow - This flow is optimized for clients that are part of a web server application, accessible via HTTP requests. This flow is described in Section 2.1 (Web Server Flow).

*User-Agent Flow - This flow is designed for clients running inside a user-agent (typically a web browser). This flow is described in Section 2.2 (User-Agent Flow).

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

*Username and Password Flow - This flow is used in cases where the end-user trusts the client to handle its credentials but it is still undesirable for the client to store the end-user's username and password. This flow is only suitable when there is a high

degree of trust between the end-user and the client. This flow is described in Section 2.3 (Username and Password Flow).

*Client Credentials Flow - The client uses its credentials to obtain an access token. This flow is described in Section 2.4 (Client Credentials Flow).

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

*Assertion Flow - The client presents an assertion such as a SAML (Cantor, S., Kemp, J., Philpott, R., and E. Maler, “Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0,” March 2005.) [OASIS.saml-core-2.0-os] assertion to the authorization server in exchange for an access token. This flow is described in Section 2.5 (Assertion Flow).

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.

1.3. Example

TOC

[[Todo]]

1.4. Notational Conventions

TOC

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] (Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels,” March 1997.).
This document uses the Augmented Backus-Naur Form (ABNF) notation of [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,” March 2010.). Additionally, the realm and auth-param rules are included from [RFC2617] (Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, “HTTP Authentication: Basic and Digest Access Authentication,” June 1999.).

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

2. Client Flows

TOC

2.1. Web Server Flow

TOC

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

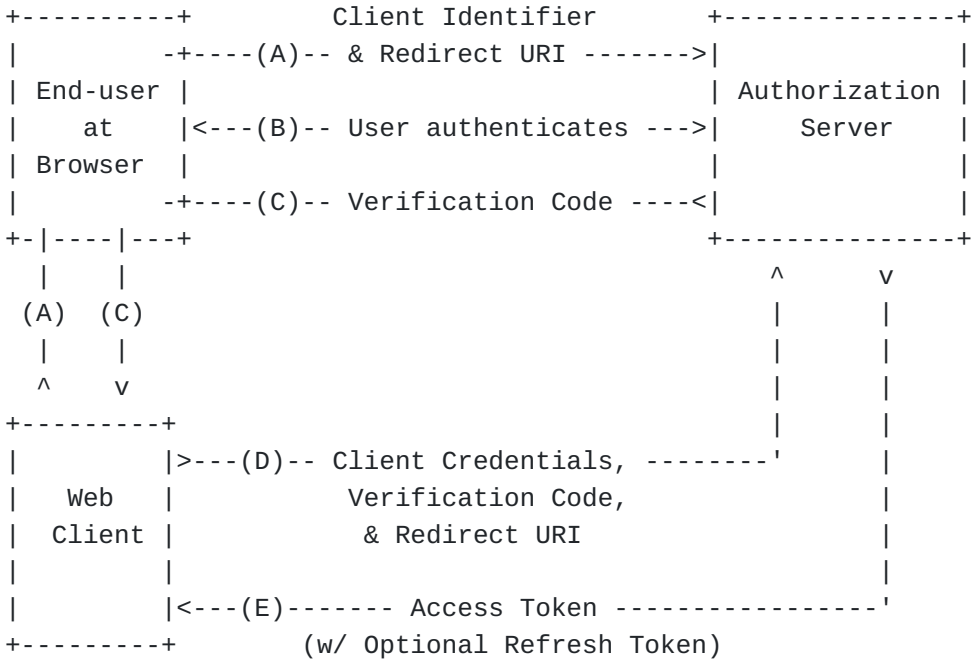

Figure 3: Web Server Flow

The web server flow illustrated in Figure 3 (Web Server Flow) 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 4.1.1 (End-User Authorization Endpoint) 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 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) Assuming the end-user granted access, the authorization server redirects the user-agent back to the client to the redirection URI provided earlier. The authorization includes a verification code for the client to use to obtain an access token.

(D) The client requests an access token from the authorization server by authenticating and including the verification code received in the previous step as described in Section 5.1 (Token Endpoint).

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

2.2. User-Agent Flow

TOC

The user-agent flow is a user delegation flow suitable for client applications residing in a user-agent, typically implemented in a browser using a scripting language such as JavaScript. These clients cannot keep client secrets confidential and the authentication of the client is based on the user-agent's same-origin policy.
Unlike other flows in which the client makes separate authorization and access token requests, the client received the access token as a result of the authorization request in the form of an HTTP redirection. The client requests the authorization server to redirect the user-agent to another web server or local resource accessible to the browser which is capable of extracting the access token from the response and passing it to the client.
This user-agent flow does not utilize the client secret since the client executables reside on the end-user's computer or device which makes the client secret accessible and exploitable. Because the access token is encoded into the redirection URI, it may be exposed to the end-user and other applications residing on the computer or device.

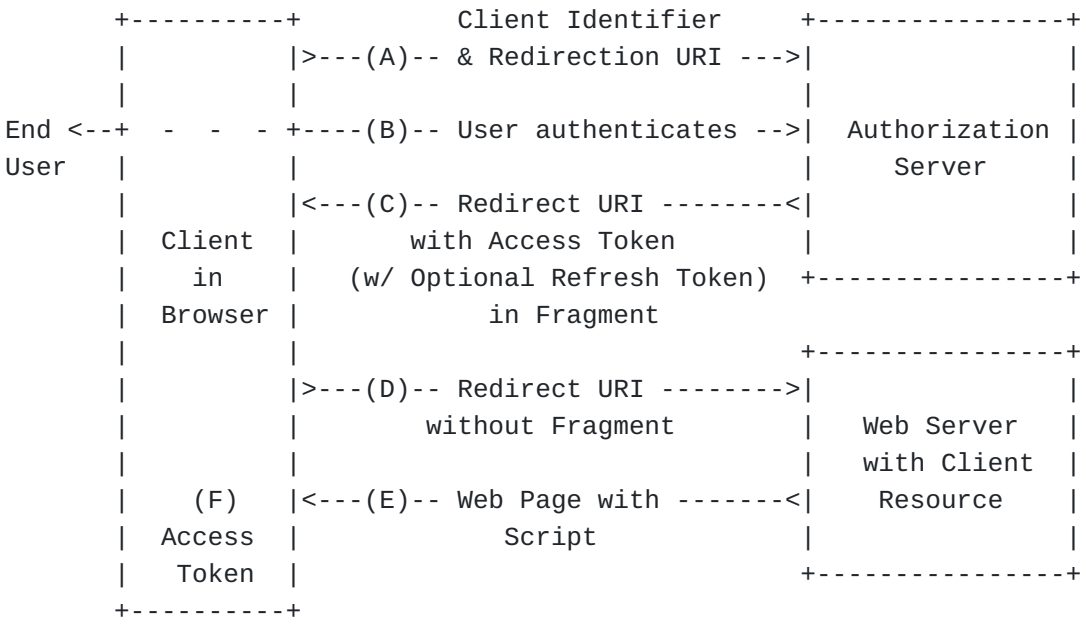

Figure 4: User-Agent Flow

The user-agent flow illustrated in Figure 4 (User-Agent Flow) includes the following steps:

(A) The client sends the user-agent to the end-user authorization endpoint as described in Section 4.1.1 (End-User Authorization Endpoint) 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) Assuming the end-user granted access, the authorization server redirects the user-agent to the redirection URI provided earlier. The redirection URI includes the access token (and an optional verification code) in the URI fragment.

(D) The user-agent follows the redirection instructions by making an HTTP GET request to the web server which does not include the fragment. The user-agent retains the fragment information locally. The user-agent MUST NOT include the fragment component with the request.

(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 a verification 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.

2.3. Username and Password Flow

TOC

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

However, unlike the HTTP Basic authentication scheme defined in [RFC2617] (Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, “HTTP Authentication: Basic and Digest Access Authentication,” June 1999.), the end-user's credentials are used in a single request and are exchanged for an access token and refresh token which eliminates the client need to store them for future use.

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

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

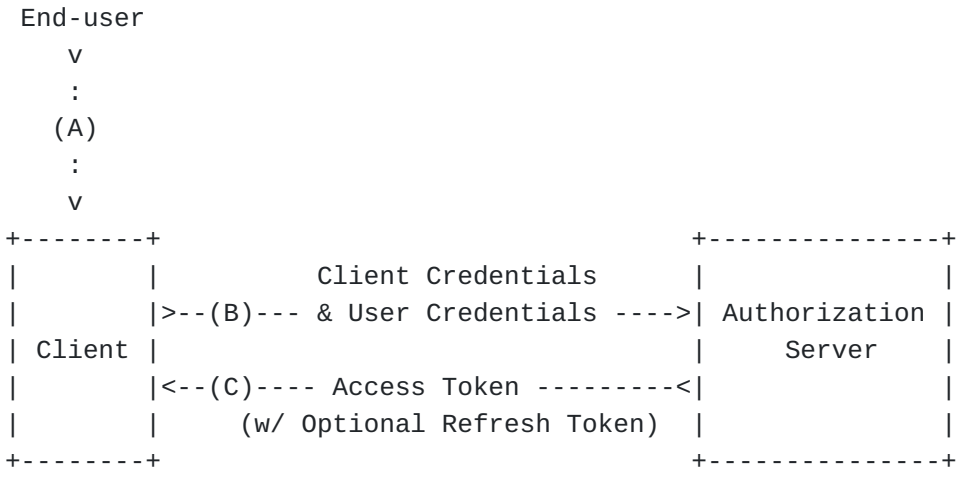

Figure 5: Username and Password Flow

The username and password flow illustrated in Figure 5 (Username and Password Flow) includes the following steps:

> **(A)** The end-user provides the client with its username and password.
>
> **(B)** The client requests an access token from the authorization server by authenticating and including the end-user's username and password, and desired scope as described in Section 5.1 (Token Endpoint).
>
> **(C)** The authorization server validates the end-user credentials and the client credentials and issues an access token.

2.4. Client Credentials Flow

TOC

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

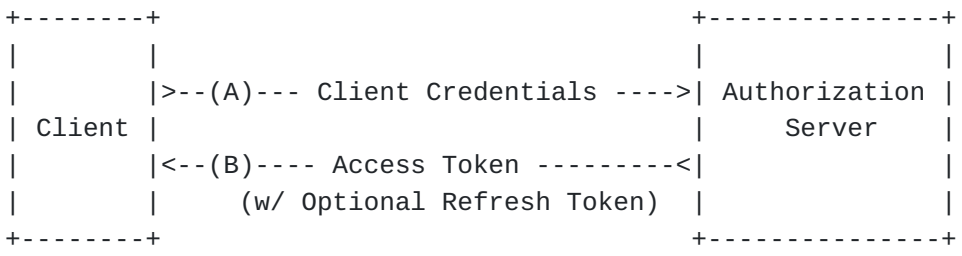

Figure 6: Client Credentials Flow

The client credential flow illustrated in Figure 6 (Client Credentials Flow) includes the following steps:

> **(A)** The client requests an access token from the authorization server by authenticating and including the desired scope as described in Section 5.1 (Token Endpoint). No additional authorization grant information is needed.
>
> **(B)** The authorization server validates the client credentials and issues an access token.

2.5. Assertion Flow

TOC

The assertion flow is used when a client wishes to exchange an existing security token or assertion for an access token. This flow is suitable when the client is the resource owner or is acting on behalf of the resource owner (based on the content of the assertion used).
The assertion flow requires the client to obtain a assertion (such as a SAML (Cantor, S., Kemp, J., Philpott, R., and E. Maler, “Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0,” March 2005.) [OASIS.saml-core-2.0-os] assertion) from an assertion issuer or to self-issue an assertion prior to initiating the flow. The assertion format, the process by which the assertion is obtained, and the method of validating the assertion are defined by the assertion issuer and the authorization server, and are beyond the scope of this specification.

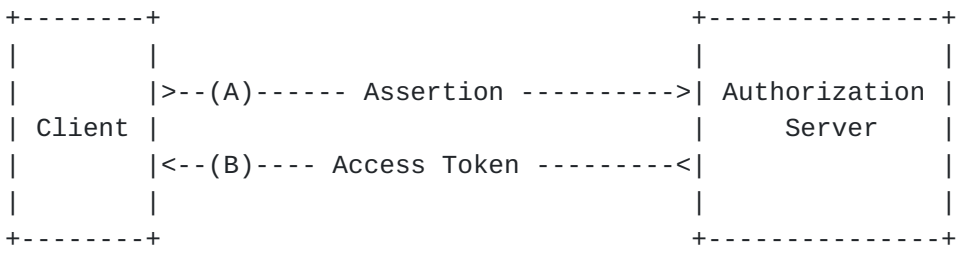

Figure 7: Assertion Flow

The assertion flow illustrated in Figure 7 (Assertion Flow) includes the following steps:

> **(A)** The client requests an access token from the authorization server by authenticating and including the assertion, assertion type, and desired scope as described in Section 5.1 (Token Endpoint).
>
> **(B)** The authorization server validates the assertion and issues an access token.

2.6. Native Application Considerations

TOC

Native application are clients running as native code on the end-user's computer or device (i.e. executing outside a browser 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 utilize many of the flows defined in this specification with little or no changes. For example:

- Launch an external user-agent and have it redirect back to the client using a custom URI scheme. This works with the web server flow and user-agent flow.

- Launch an external user-agent and poll for changes to the window title. This works with the web server flow with a server-hosted custom redirect result page that puts the verification code in the title.

*Use an embedded user-agent and obtain the redirection URI. This works with the web server flow and user-agent flow.

*Use the username and password flow and prompt the end-users for their credentials. This is generally discouraged as it hands the end-user's password directly to the 3rd party and may not work with some authentication schemes.

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

---

3. Client Credentials

TOC

When requesting access from the authorization server, the client identifies itself using a set of client credentials. The client credentials include a client identifier and an OPTIONAL symmetric shared secret. The means through which the client obtains these credentials are beyond the scope of this specification, but usually involve registration with the authorization server.
The client identifier is used by the authorization server to establish the identity of the client for the purpose of presenting information to the resource owner prior to granting access, as well as for providing different service levels to different clients. They can also be used to block unauthorized clients from requesting access.
Due to the nature of some clients, authorization servers SHOULD NOT make assumptions about the confidentiality of client credentials without establishing trust with the client operator. Authorization servers SHOULD NOT issue client secrets to clients incapable of keeping their secrets confidential.

TOC

3.1. Client Authentication

The token endpoint requires the client to authenticate itself to the authorization server. This is done by including the client identifier (and optional secret) in the request. The client identifier and secret are included in the request using two request parameters: client_id and client_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=gX1fBat3bV&code=i1WsRn1uB1&
redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

The client MAY include the client credentials using an HTTP authentication scheme which supports authenticating using a username and password, instead of using the client_id and client_secret request parameters. Including the client credentials using an HTTP authentication scheme fulfills the requirements of including the parameters as defined by the various flows.
The client MUST NOT include the client credentials using more than one mechanism. If more than one mechanism is used, regardless if the credentials are identical, the server MUST reply with an HTTP 400 status code (Bad Request) and include the multiple-credentials error message.
The authorization server MUST accept the client credentials using both the request parameters, and the HTTP Basic authentication scheme as defined in [RFC2617] (Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, “HTTP Authentication: Basic and Digest Access Authentication,” June 1999.).
The authorization server MAY support additional HTTP authentication schemes.
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=i1WsRn1uB1&
redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

4. Establishing Resource Owner Authorization

TOC

Before the client can obtain an access token, it must first attain authorization from the resource owner. The methods through which the client attains authorization are codified in the various authorization flows defined in Section 5 (Obtaining an Access Token), and depends on the client type and its trust relationship with the resource owner. Resource owner authorization can be expressed in multiple ways: a verification code obtained through direct interaction with an end-user, the resource owner credentials (or the client credentials when the client is also the resource owner) obtained through a trust relationship with the resource owner, or an assertion obtained through means beyond the scope of this specification.

4.1. Verification Code

TOC

When an end-user is involved, the client attains authorization in the form of a verification code by sending the end-user to the authorization server to review and grant the request. The client sends the end-user by directing the end-user's user-agent to the authorization server's end-user authorization endpoint.

4.1.1. End-User Authorization Endpoint

TOC

When directed to 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] (Berners-Lee, T., Fielding, R., and L. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” January 2005.) 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] (Hors, A., Raggett, D., and I. Jacobs, "HTML 4.01 Specification," December 1999.):

> **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 3 (Client Credentials).
>
> **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] (Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," January 2005.) 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.

4.1.1.1. Authorization Server Response

TOC

If the end-user grants the access request, the authorization server issues an access token, a verification 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 verification code generated by the authorization server. The verification code SHOULD expire shortly after it is issued and allowed for a single use. The verification 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] (Hors, A., Raggett, D., and I. Jacobs, "HTML 4.01 Specification," December 1999.).
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 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] (Hors, A., Raggett, D., and I. Jacobs, "HTML 4.01 Specification," December 1999.). [[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
```

4.2. Resource Owner Credentials

TOC

While OAuth seeks to eliminate the need for resource owners to share their credentials with the client, possesion of the resource owner credentials constitute an authorization grant (if supported by the authorization server). Resource owner credentials should only be used when there is a high degree of trust between the resource owner the client.
In cases where the client is also the resource owner, the client credentials can be used to obtain an access token provisioned for accessing the client's protected resources.

4.3. Assertion

TOC

Assertions enable the client to utilize existing trust relationships or different authorization constructs to obtain an access token. They provide a bridge between OAuth and other trust frameworks. The authorization 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.

5. Obtaining an Access Token

TOC

The client obtains an access token by authenticating with the authorization server and presenting its authorization grant.
In many cases it is desirable to issue access tokens with a shorter lifetime than the duration of the authorization grant. However, it may be undesirable to require the resource owner to authorize the request again. Instead, the authorization server issues a refresh token in addition to the access token. When the access token expires, the client can request a new access token without involving the resource owner as long as the authorization grant is still valid. The token refresh method is described in Section 5.1.4 (Refresh Token).

5.1. Token Endpoint

TOC

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 authorization grant (in the form of a verification 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] (Dierks, T. and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,”

August 2008.) and MAY support addition mechanisms with equivalent protections.
The client obtains an access token by constructing a token request. The client constructs the request URI by:

*Adding its client credentials to the request as described in Section 3.1 (Client Authentication). For example, if the client uses a set of basic client credentials, it adds the client_id and client_secret parameters to the request (or uses the HTTP Basic authentication scheme).

*Adding the authorization grand in the form of a verification code, resource owner credentials, an assertion, or refresh token. If the client is acting on behalf of itself (the client is also the resource owner), no additional information is needed. The authorization grant is added to the request URI query component using the application/x-www-form-urlencoded format as described below.

5.1.1. Verification Code

TOC

The client includes the verification code using following parameters:

> **code** REQUIRED. The verification 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

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

The authorization server MUST verify that the verification code, client identity, client secret, and redirection URI are all valid and match its stored association. If the request is valid, the authorization

server issues a successful response as described in Section 5.1.5 (Access Token Response).

5.1.2. Resource Owner Credentials

TOC

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

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

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

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 5.1.5 (Access Token Response).
If the client is acting on behalf of itself (the client is also the resource owner), the client authentication alone suffice and the username and password parameters MUST NOT be used.

5.1.3. Assertion

TOC

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.

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

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

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

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

The authorization server MUST validate the assertion and if valid issues an access token response as described in Section 5.1.5 (Access Token Response). 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.

5.1.4. Refresh Token

TOC

Token refresh is used when the lifetime of an access token is shorter than the lifetime of the authorization grant. It enables the client to obtain a new access token without having to go through the authorization flow again or involve the resource owner.
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

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 5.1.5 (Access Token Response). 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.

5.1.5. Access Token Response

TOC

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 5.1.4 (Refresh Token).
>
> **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] (Crockford, D., “The application/json Media Type for JavaScript Object

Notation (JSON)," July 2006.). 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"
}
```

5.1.6. Error Response

TOC

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 5.1.6.1 (Error Codes).

For example:

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

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

TOC

5.1.6.1. Error Codes

[[expalain each error code:]]

*redirect_uri_mismatch

*bad_verification_code

*incorrect_client_credentials

*unauthorized_client' - The client is not permitted to use this authorization grant type.

*invalid_assertion

*unknown_format

*authorization_expired

---

6. Accessing a Protected Resource

TOC

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 as described in Section 7.1 (The WWW-Authenticate Response Header).
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 6.1 (The Authorization Request Header). Alternatively, clients MAY include the access token using the HTTP request URI in the query component as described in Section 6.2 (URI Query Parameter), or in the HTTP body when using the application/x-www-form-urlencoded content type as described in Section 6.3 (Form-Encoded Body Parameter).
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]]

6.1. The Authorization Request Header

TOC

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] (Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, “HTTP Authentication: Basic and Digest Access Authentication,” June 1999.) as follows:

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

6.2. URI Query Parameter

TOC

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] (Berners-Lee, T., Fielding, R., and L. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” January 2005.) using the oauth_token parameter.
For example, the client makes the following HTTPS request:

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

The HTTP request URI query can include other request-specific parameters, in which case, the oauth_token parameters SHOULD be appended following the request-specific parameters, properly separated by an & character (ASCII code 38).
The resource server MUST validate the access token and ensure it has not expired and its scope includes the requested resource. If the resource expired or is not valid, the resource server MUST reply with an HTTP 401 status code (Unauthorized) and include the HTTP WWW-Authenticate response header as described in Section 7.1 (The WWW-Authenticate Response Header).

6.3. Form-Encoded Body Parameter

TOC

When including the access token in the HTTP request entity-body, the client adds the access token to the request body using the oauth_token parameter. The client can use this method only if the following REQUIRED conditions are met:

- The entity-body is single-part.
- The entity-body follows the encoding requirements of the application/x-www-form-urlencoded content-type as defined by [W3C.REC-html401-19991224] (Hors, A., Raggett, D., and I. Jacobs, “HTML 4.01 Specification,” December 1999.).
- The HTTP request entity-header includes the Content-Type header field set to application/x-www-form-urlencoded.
- The HTTP request method is POST, PUT, or DELETE.

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

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

oauth_token=vF9dft4qmT
```

The resource server MUST validate the access token and ensure it has not expired and its scope includes the requested resource. If the resource expired or is not valid, the resource server MUST reply with an HTTP 401 status code (Unauthorized) and include the HTTP WWW-Authenticate response header as described in Section 7.1 (The WWW-Authenticate Response Header).

---

7. Identifying a Protected Resource

TOC

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.

7.1. The WWW-Authenticate Response Header

TOC

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] (Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, “HTTP Authentication: Basic and Digest Access Authentication,” June 1999.) 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] (Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication," June 1999.).
The error attribute is used to inform the client the reason why an access request was declined. [[Add list of error codes]]

---

8. Security Considerations

TOC

[[Todo]]

---

9. IANA Considerations

TOC

[[Not Yet]]

Appendix A. Contributors

TOC

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 B. Acknowledgements

TOC

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

Appendix C. Document History

TOC

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

-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

TOC

10.1. Normative References

TOC

[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 (TXT).
[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 (TXT).
[RFC2104]	Krawczyk, H., Bellare, M., and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication,” RFC 2104, February 1997 (TXT).
[RFC2119]	Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels,” BCP 14, RFC 2119, March 1997 (TXT, HTML, XML).
[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 (TXT, PS, PDF, HTML, XML).
[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 (TXT, HTML, XML).
[RFC2818]	Rescorla, E., “HTTP Over TLS,” RFC 2818, May 2000 (TXT).
[RFC3023]	Murata, M., St. Laurent, S., and D. Kohn, “XML Media Types,” RFC 3023, January 2001 (TXT).
[RFC3447]	Jonsson, J. and B. Kaliski, “Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1,” RFC 3447, February 2003 (TXT).
[RFC3629]	Yergeau, F., “UTF-8, a transformation format of ISO 10646,” STD 63, RFC 3629, November 2003 (TXT).
[RFC3986]	Berners-Lee, T., Fielding, R., and L. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” STD 66, RFC 3986, January 2005 (TXT, HTML, XML).
[RFC4627]	Crockford, D., “The application/json Media Type for JavaScript Object Notation (JSON),” RFC 4627, July 2006 (TXT).
[RFC5246]	Dierks, T. and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” RFC 5246, August 2008 (TXT).
[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 (HTML).

10.2. Informative References

TOC

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

TOC

	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/