Network Working Group	E. Hammer, Ed.
Internet-Draft	
Obsoletes: 5849 (if approved)	D. Recordon
Intended status: Standards Track	Facebook
Expires: July 24, 2012	D. Hardt
	Microsoft
	January 21, 2012

TOC

The OAuth 2.0 Authorization Protocol draft-ietf-oauth-v2-23

Abstract

The OAuth 2.0 authorization protocol enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. This specification replaces and obsoletes the OAuth 1.0 protocol described in RFC 5849.

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 July 24, 2012.

Copyright Notice

Copyright (c) 2012 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. Roles
  - 1.2. Protocol Flow
  - 1.3. Authorization Grant
    - 1.3.1. Authorization Code
    - 1.3.2. Implicit
    - 1.3.3. Resource Owner Password Credentials
    - 1.3.4. Client Credentials
  - 1.4. Access Token
  - 1.5. Refresh Token

1.6. TLS Version
1.7. Interoperability
1.8. Notational Conventions
2. Client Registration
2.1. Client Types
2.2. Client Identifier
2.3. Client Authentication
2.3.1. Client Password
2.3.2. Other Authentication Methods
2.4. Unregistered Clients
3. Protocol Endpoints
3.1. Authorization Endpoint
3.1.1. Response Type
3.1.2. Redirection Endpoint
3.2. Token Endpoint
3.2.1. Client Authentication
3.3. Access Token Scope
4. Obtaining Authorization
4.1. Authorization Code Grant
4.1.1. Authorization Request
4.1.2. Authorization Response
4.1.3. Access Token Request
4.1.4. Access Token Response
4.2. Implicit Grant
4.2.1. Authorization Request
4.2.2. Access Token Response
4.3. Resource Owner Password Credentials Grant
4.3.1. Authorization Request and Response
4.3.2. Access Token Request
4.3.3. Access Token Response
4.4. Client Credentials Grant
4.4.1. Authorization Request and Response
4.4.2. Access Token Request
4.4.3. Access Token Response
4.5. Extension Grants
5. Issuing an Access Token
5.1. Successful Response
5.2. Error Response
6. Refreshing an Access Token
7. Accessing Protected Resources
7.1. Access Token Types
8. Extensibility
8.1. Defining Access Token Types
8.2. Defining New Endpoint Parameters
8.3. Defining New Authorization Grant Types
8.4. Defining New Authorization Endpoint Response Types
8.5. Defining Additional Error Codes
9. Native Applications
10. Security Considerations
10.1. Client Authentication
10.2. Client Impersonation
10.3. Access Tokens
10.4. Refresh Tokens
10.5. Authorization Codes
10.6. Authorization Code Redirection URI Manipulation
10.7. Resource Owner Password Credentials
10.8. Request Confidentiality
10.9. Endpoints Authenticity
10.10. Credentials Guessing Attacks
10.11. Phishing Attacks
10.12. Cross-Site Request Forgery
10.13. Clickjacking
10.14. Code Injection and Input Validation
10.15. Open Redirectors
11. IANA Considerations
11.1. The OAuth Access Token Type Registry
11.1.1. Registration Template
11.2. The OAuth Parameters Registry

11.2.1. Registration Template
11.2.2. Initial Registry Contents
11.3. The OAuth Authorization Endpoint Response Type Registry
11.3.1. Registration Template
11.3.2. Initial Registry Contents
11.4. The OAuth Extensions Error Registry
11.4.1. Registration Template
12. Acknowledgements
Appendix A. Editor's Notes
13. References
13.1. Normative References
13.2. Informative References
§ Authors' Addresses

---

1. Introduction

TOC

In the traditional client-server authentication model, the client requests an access restricted resource (protected resource) on the server by authenticating with the server using the resource owner's credentials. In order to provide third-party applications access to restricted resources, the resource owner shares its credentials with the third-party. This creates several problems and limitations:

- Third-party applications are required to store the resource owner's credentials for future use, typically a password in clear-text.
- Servers are required to support password authentication, despite the security weaknesses created by passwords.
- Third-party applications gain overly broad access to the resource owner's protected resources, leaving resource owners without any ability to restrict duration or access to a limited subset of resources.
- Resource owners cannot revoke access to an individual third-party without revoking access to all third-parties, and must do so by changing their password.
- Compromise of any third-party application results in compromise of the end-user s password and all of the data protected by that password.

OAuth addresses these issues by introducing an authorization layer and separating the role of the client from that of the resource owner. In OAuth, the client requests access to resources controlled by the resource owner and hosted by the resource server, and is issued a different set of credentials than those of the resource owner.

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

For example, an end-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 a server trusted by the photo sharing service (authorization server) which issues the printing service delegation-specific credentials (access token).

This specification is designed for use with HTTP **[RFC2616]**. The use of OAuth with any transport protocol other than HTTP is undefined.

1.1. Roles

TOC

OAuth defines four roles:

resource owner
: An entity capable of granting access to a protected resource (e.g. end-user).

resource server
: The server hosting the protected resources, capable of accepting and responding

to protected resource requests using access tokens.

client
: An application making protected resource requests on behalf of the resource owner and with its authorization. The term client does not imply any particular implementation characteristics (e.g. whether the application executes on a server, a desktop, or other devices).

authorization server
: The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

The interaction between the authorization server and resource server is beyond the scope of this specification. The authorization server may be the same server as the resource server or a separate entity. A single authorization server may issue access tokens accepted by multiple resource servers.

1.2. Protocol Flow

TOC

```
+--------+                               +---------------+
|        |--(A)- Authorization Request ->|   Resource    |
|        |                               |     Owner     |
|        |<-(B)-- Authorization Grant ---|               |
|        |                               +---------------+
|        |
|        |                               +---------------+
|        |--(C)-- Authorization Grant -->| Authorization |
| Client |                               |     Server    |
|        |<-(D)----- Access Token -------|               |
|        |                               +---------------+
|        |
|        |                               +---------------+
|        |--(E)----- Access Token ------>|    Resource   |
|        |                               |     Server    |
|        |<-(F)--- Protected Resource ---|               |
+--------+                               +---------------+
```

Figure 1: Abstract Protocol Flow

The abstract flow illustrated in **Figure 1** describes the interaction between the four roles and includes the following steps:

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

(B)
: The client receives an authorization grant which is a credential representing the resource owner's authorization, expressed using one of four grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.

(C)
: The client requests an access token by authenticating with the authorization server and presenting the authorization grant.

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

(E)
: The client requests the protected resource from the resource server and authenticates by presenting the access token.

(F)

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

TOC

1.3. Authorization Grant

An authorization grant is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token. This specification defines four grant types: authorization code, implicit, resource owner password credentials, and client credentials, as well as an extensibility mechanism for defining additional types.

TOC

1.3.1. Authorization Code

The authorization code is obtained by using an authorization server as an intermediary between the client and resource owner. Instead of requesting authorization directly from the resource owner, the client directs the resource owner to an authorization server (via its user-agent as defined in **[RFC2616]**), which in turn directs the resource owner back to the client with the authorization code.

Before directing the resource owner back to the client with the authorization code, the authorization server authenticates the resource owner and obtains authorization. Because the resource owner only authenticates with the authorization server, the resource owner's credentials are never shared with the client.

The authorization code provides a few important security benefits such as the ability to authenticate the client, and the transmission of the access token directly to the client without passing it through the resource owner's user-agent, potentially exposing it to others, including the resource owner.

TOC

1.3.2. Implicit

The implicit grant is a simplified authorization code flow optimized for clients implemented in a browser using a scripting language such as JavaScript. In the implicit flow, instead of issuing the client an authorization code, the client is issued an access token directly (as the result of the resource owner authorization). The grant type is implicit as no intermediate credentials (such as an authorization code) are issued (and later used to obtain an access token).

When issuing an access token during the implicit grant flow, the authorization server does not authenticate the client. In some cases, the client identity can be verified via the redirection URI used to deliver the access token to the client. The access token may be exposed to the resource owner or other applications with access to the resource owner's user-agent.

Implicit grants improve the responsiveness and efficiency of some clients (such as a client implemented as an in-browser application) since it reduces the number of round trips required to obtain an access token. However, this convenience should be weighed against the security implications of using implicit grants, especially when the authorization code grant type is available.

TOC

1.3.3. Resource Owner Password Credentials

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

Even though this grant type requires direct client access to the resource owner credentials, the resource owner credentials are used for a single request and are exchanged for an access token. This grant type can eliminate the need for the client to store the resource

owner credentials for future use, by exchanging the credentials with a long-lived access token or refresh token.

1.3.4. Client Credentials

TOC

The client credentials (or other forms of client authentication) can be used as an authorization grant when the authorization scope is limited to the protected resources under the control of the client, or to protected resources previously arranged with the authorization server. Client credentials are used as an authorization grant typically when the client is acting on its own behalf (the client is also the resource owner), or is requesting access to protected resources based on an authorization previously arranged with the authorization server.

1.4. Access Token

TOC

Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server.

The token may denote an identifier used to retrieve the authorization information, or self-contain the authorization information in a verifiable manner (i.e. a token string consisting of some data and a signature). Additional authentication credentials, which are beyond the scope of this specification, may be required in order for the client to use a token.

The access token provides an abstraction layer, replacing different authorization constructs (e.g. username and password) with a single token understood by the resource server. This abstraction enables issuing access tokens more restrictive than the authorization grant used to obtain them, as well as removing the resource server's need to understand a wide range of authentication methods.

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

1.5. Refresh Token

TOC

Refresh tokens are credentials used to obtain access tokens. Refresh tokens are issued to the client by the authorization server and are used to obtain a new access token when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope (access tokens may have a shorter lifetime and fewer permissions than authorized by the resource owner). Issuing a refresh token is optional at the discretion of the authorization server. If the authorization server issues a refresh token, it is included when issuing an access token.

A refresh token is a string representing the authorization granted to the client by the resource owner. The string is usually opaque to the client. The token denotes an identifier used to retrieve the authorization information. Unlike access tokens, refresh tokens are intended for use only with authorization servers and are never sent to resource servers.

```
+--------+                                    +---------------+
|        |--(A)------- Authorization Grant --------->|               |
|        |                                    |               |
|        |<-(B)----------- Access Token -------------|               |
|        |               & Refresh Token      |               |
|        |                                    |               |
```

```
  |         |                              +----------+   |                |
  |         |--(C)---- Access Token ---->|          |   |                |
  |         |                              |          |   |                |
  |         |<-(D)- Protected Resource --| Resource |   | Authorization |
  | Client |                              |  Server  |   |     Server     |
  |         |--(E)---- Access Token ---->|          |   |                |
  |         |                              |          |   |                |
  |         |<-(F)- Invalid Token Error -|          |   |                |
  |         |                              +----------+   |                |
  |         |                                             |                |
  |         |--(G)----------- Refresh Token ----------->|                |
  |         |                                             |                |
  |         |<-(H)----------- Access Token -------------|                |
 +--------+           & Optional Refresh Token        +---------------+
```

Figure 2: Refreshing an Expired Access Token

The flow illustrated in **Figure 2** includes the following steps:

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

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

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

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

(E)
: Steps (C) and (D) repeat until the access token expires. If the client knows the access token expired, it skips to step (G), otherwise it makes another protected resource request.

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

(G)
: The client requests a new access token by authenticating with the authorization server and presenting the refresh token. The client authentication requirements are based on the client type and on the authorization server policies.

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

Steps C, D, E, and F are outside the scope of this specification as described in **Section 7**.

1.6. TLS Version

TOC

Whenever TLS is required by this specification, the appropriate version (or versions) of TLS will vary over time, based on the widespread deployment and known security vulnerabilities. At the time of this writing, TLS version 1.2 **[RFC5246]** is the most recent version, but has a very limited deployment base and might not be readily available for implementation. TLS version 1.0 **[RFC2246]** is the most widely deployed version, and will provide the broadest interoperability.

Implementations MAY also support additional transport-layer mechanisms that meet their security requirements.

1.7. Interoperability

TOC

OAuth 2.0 provides a rich authorization framework with well-defined security properties. However, as a rich and highly extensible framework with many optional components, this specification is likely to produce a wide range of non-interoperable implementations. In addition, this specification leaves a few required components partially or fully undefined (e.g. client registration, authorization server capabilities, endpoint discovery).

This protocol was design with the clear expectation that future work will define prescriptive profiles and extensions necessary to achieve full web-scale interoperability.

1.8. Notational Conventions

TOC

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

This specification uses the Augmented Backus-Naur Form (ABNF) notation of **[RFC5234]**.

Certain security-related terms are to be understood in the sense defined in **[RFC4949]**. These terms include, but are not limited to, 'attack', 'authentication', 'authorization', 'certificate', 'confidentiality', 'credential', 'encryption', 'identity', 'sign', 'signature', 'trust', 'validate', and 'verify'.

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

---

2. Client Registration

TOC

Before initiating the protocol, the client registers with the authorization server. The means through which the client registers with the authorization server are beyond the scope of this specification, but typically involve end-user interaction with an HTML registration form.

Client registration does not require a direct interaction between the client and the authorization server. When supported by the authorization server, registration can rely on other means for establishing trust and obtaining the required client properties (e.g. redirection URI, client type). For example, registration can be accomplished using a self-issued or third-party-issued assertion, or by the authorization server performing client discovery using a trusted channel.

When registering a client, the client developer:

- specifies the client type as described in **Section 2.1**,
- provides its client redirection URIs as described in **Section 3.1.2**, and
- includes any other information required by the authorization server (e.g. application name, website, description, logo image, the acceptance of legal terms).

2.1. Client Types

TOC

OAuth defines two client types, based on their ability to authenticate securely with the authorization server (i.e. ability to maintain the confidentiality of their client credentials):

confidential
: Clients capable of maintaining the confidentiality of their credentials (e.g. client implemented on a secure server with restricted access to the client credentials), or capable of secure client authentication using other means.

public
: Clients incapable of maintaining the confidentiality of their credentials (e.g. clients executing on the device used by the resource owner such as an installed native application or a web browser-based application), and incapable of secure client authentication via any other means.

The client type designation is based on the authorization server's definition of secure

authentication and its acceptable exposure levels of client credentials.

The authorization server SHOULD NOT make assumptions about the client type, nor accept the type information provided by the client developer without first establishing trust.

A client application consisting of multiple components, each with its own client type (e.g. a distributed client with both a confidential server-based component and a public browser-based component), MUST register each component separately as a different client to ensure proper handling by the authorization server. The authorization server MAY provider tools to manage such complex clients through a single administration interface.

This specification has been designed around the following client profiles:

web application
: A web application is a confidential client running on a web server. Resource owners access the client via an HTML user interface rendered in a user-agent on the device used by the resource owner. The client credentials as well as any access token issued to the client are stored on the web server and are not exposed to or accessible by the resource owner.

user-agent-based application
: A user-agent-based application is a public client in which the client code is downloaded from a web server and executes within a user-agent (e.g. web browser) on the device used by the resource owner. Protocol data and credentials are easily accessible (and often visible) to the resource owner. Since such applications reside within the user-agent, they can make seamless use of the user-agent capabilities when requesting authorization.

native application
: A native application is a public client installed and executed on the device used by the resource owner. Protocol data and credentials are accessible to the resource owner. It is assumed that any client authentication credentials included in the application can be extracted. On the other hand, dynamically issued credentials such as access tokens or refresh tokens can receive an acceptable level of protection. At a minimum, these credentials are protected from hostile servers with which the application may interact with. On some platforms these credentials might be protected from other applications residing on the same device.

TOC

2.2. Client Identifier

The authorization server issues the registered client a client identifier - a unique string representing the registration information provided by the client. The client identifier is not a secret, it is exposed to the resource owner, and MUST NOT be used alone for client authentication.

TOC

2.3. Client Authentication

If the client type is confidential, the client and authorization server establish a client authentication method suitable for the security requirements of the authorization server. The authorization server MAY accept any form of client authentication meeting its security requirements.

Confidential clients are typically issued (or establish) a set of client credentials used for authenticating with the authorization server (e.g. password, public/private key pair).

The authorization server MAY establish a client authentication method with public clients. However, the authorization server MUST NOT rely on public client authentication for the purpose of identifying the client.

The client MUST NOT use more than one authentication method in each request.

TOC

2.3.1. Client Password

Clients in possession of a client password MAY use the HTTP Basic authentication scheme as defined in **[RFC2617]** to authenticate with the authorization server. The client identifier is used as the username, and the client password is used as the password. The authorization server MUST support the HTTP Basic authentication scheme for authenticating clients which were issued a client password.

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

```
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
```

Alternatively, the authorization server MAY support including the client credentials in the request body using the following parameters:

client_id
: REQUIRED. The client identifier issued to the client during the registration process described by **Section 2.2**.

client_secret
: REQUIRED. The client secret. The client MAY omit the parameter if the client secret is an empty string.

Including the client credentials in the request body using the two parameters is NOT RECOMMENDED, and should be limited to clients unable to directly utilize the HTTP Basic authentication scheme (or other password-based HTTP authentication schemes). The parameters can only be transmitted in the request body and MUST NOT be included in the request URI.

For example, requesting to refresh an access token (**Section 6**) using the body parameters (extra line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded;charset=UTF-8

grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
&client_id=s6BhdRkqt3&client_secret=7Fjfp0ZBr1KtDRbnfVdmIw
```

The authorization server MUST require TLS as described in **Section 1.6** when sending requests using password authentication.

Since this client authentication method involves a password, the authorization server MUST protect any endpoint utilizing it against brute force attacks.

2.3.2. Other Authentication Methods

TOC

The authorization server MAY support any suitable HTTP authentication scheme matching its security requirements. When using other authentication methods, the authorization server MUST define a mapping between the client identifier (registration record) and authentication scheme.

2.4. Unregistered Clients

TOC

This specification does not exclude the use of unregistered clients. However, the use with such clients is beyond the scope of this specification, and requires additional security analysis and review of its interoperability impact.

3. Protocol Endpoints

TOC

The authorization process utilizes two authorization server endpoints (HTTP resources):

- Authorization endpoint - used to obtain authorization from the resource owner via user-agent redirection.
- Token endpoint - used to exchange an authorization grant for an access token, typically with client authentication.

As well as one client endpoint:

- Redirection endpoint - used to return authorization credentials responses from the authorization server to the client via the resource owner user-agent.

Not every authorization grant type utilizes both endpoints. Extension grant types MAY define additional endpoints as needed.

3.1. Authorization Endpoint

TOC

The authorization endpoint is used to interact with the resource owner and obtain an authorization grant. The authorization server MUST first verify the identity of the resource owner. The way in which the authorization server authenticates the resource owner (e.g. username and password login, session cookies) is beyond the scope of this specification.

The means through which the client obtains the location of the authorization endpoint are beyond the scope of this specification, but the location is typically provided in the service documentation.

The endpoint URI MAY include an `application/x-www-form-urlencoded` formatted (**[W3C.REC-html401-19991224]**) query component (**[RFC3986]** section 3.4), which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

Since requests to the authorization endpoint result in user authentication and the transmission of clear-text credentials (in the HTTP response), the authorization server MUST require TLS as described in **Section 1.6** when sending requests to the authorization endpoint.

The authorization server MUST support the use of the HTTP `GET` method **[RFC2616]** for the authorization endpoint, and MAY support the use of the `POST` method as well.

Parameters sent without a value MUST be treated as if they were omitted from the request. The authorization server MUST ignore unrecognized request parameters. Request and response parameters MUST NOT be included more than once.

3.1.1. Response Type

TOC

The authorization endpoint is used by the authorization code grant type and implicit grant type flows. The client informs the authorization server of the desired grant type using the following parameter:

response_type
: REQUIRED. The value MUST be one of `code` for requesting an authorization code as described by **Section 4.1.1**, `token` for requesting an access token (implicit grant) as described by **Section 4.2.1**, or a registered extension value as described by **Section 8.4**.

Extension response types MAY contain a space-delimited (%x20) list of values, where the order of values does not matter (e.g. response type `a b` is the same as `b a`). The meaning of such composite response types is defined by their respective specifications.

If an authorization request is missing the `response_type` parameter, the authorization server MUST return an error response as described in **Section 4.1.2.1**.

3.1.2. Redirection Endpoint

TOC

After completing its interaction with the resource owner, the authorization server directs the resource owner's user-agent back to the client. The authorization server redirects the user-agent to the client's redirection endpoint previously established with the authorization server during the client registration process or when making the authorization request.

The redirection endpoint URI MUST be an absolute URI as defined by **[RFC3986]** section 4.3. The endpoint URI MAY include an `application/x-www-form-urlencoded` formatted (**[W3C.REC-html401-19991224]**) query component (**[RFC3986]** section 3.4), which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

3.1.2.1. Endpoint Request Confidentiality

TOC

The redirection endpoint SHOULD require the use of TLS as described in **Section 1.6** when the requested response type is `code` or `token`, or when the redirection request will result in the transmission of sensitive credentials over an open network. This specification does not mandate the use of TLS because at the time of this writing, requiring clients to deploy TLS is a significant hurdle for most client developers.

Lack of transport-layer security can have a severe impact on the security of the client and the protected resources it is authorized to access. The use of transport-layer security is particularly critical when the authorization process is used as a form of delegated end-user authentication by the client (e.g. third-party sign-in service).

3.1.2.2. Registration Requirements

TOC

The authorization server MUST require the following clients to register their redirection endpoint:

- Public clients.
- Confidential clients utilizing the implicit grant type.

The authorization server SHOULD require all clients to register their redirection endpoint prior to utilizing the authorization endpoint

The authorization server SHOULD require the client to provide the complete redirection URI (the client MAY use the `state` request parameter to achieve per-request customization). If requiring the registration of the complete redirection URI is not possible, the authorization server SHOULD require the registration of the URI scheme, authority, and path (allowing the client to dynamically vary only the query component of the redirection URI when requesting authorization).

The authorization server MAY allow the client to register multiple redirection endpoints.

Lack of a redirection URI registration requirement can enable an attacker to use the authorization endpoint as open redirector as described in **Section 10.15**.

3.1.2.3. Dynamic Configuration

TOC

If multiple redirection URIs have been registered, if only part of the redirection URI has been registered, or if no redirection URI has been registered, the client MUST include a redirection URI with the authorization request using the `redirect_uri` request parameter.

When a redirection URI is included in an authorization request, the authorization server MUST compare and match the value received against at least one of the registered redirection URIs (or URI components) as defined in **[RFC3986]** section 6, if any redirection URIs were registered. If the client registration included the full redirection URI, the authorization server MUST compare the two URIs using simple string comparison as defined in **[RFC3986]** section 6.2.1.

3.1.2.4. Invalid Endpoint

TOC

If an authorization request fails validation due to a missing, invalid, or mismatching redirection URI, the authorization server SHOULD inform the resource owner of the error, and MUST NOT automatically redirect the user-agent to the invalid redirection URI.

3.1.2.5. Endpoint Content

TOC

The redirection request to the client's endpoint typically results in an HTML document response, processed by the user-agent. If the HTML response is served directly as the result of the redirection request, any script included in the HTML document will execute with full access to the redirection URI and the credentials it contains.

The client MUST NOT include any untrusted third-party scripts in the redirection endpoint response (e.g. third-party analytics, social plug-ins, ad networks) without first ensuring that its own scripts used to extract and remove the credentials from the URI will execute first.

The client SHOULD NOT include any third-party scripts in the redirection endpoint response. Instead, it should extract the credentials from the URI and redirect the user-agent again to another endpoint without the credentials in the URI.

3.2. Token Endpoint

TOC

The token endpoint is used by the client to obtain an access token by presenting its authorization grant or refresh token. The token endpoint is used with every authorization grant except for the implicit grant type (since an access token is issued directly).

The means through which the client obtains the location of the token endpoint are beyond the scope of this specification but is typically provided in the service documentation.

The endpoint URI MAY include an `application/x-www-form-urlencoded` formatted (**[W3C.REC-html401-19991224]**) query component (**[RFC3986]** section 3.4), which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

Since requests to the token endpoint result in the transmission of clear-text credentials (in the HTTP request and response), the authorization server MUST require TLS as described in **Section 1.6** when sending requests to the token endpoint.

The client MUST use the HTTP `POST` method when making access token requests.

Parameters sent without a value MUST be treated as if they were omitted from the request. The authorization server MUST ignore unrecognized request parameters. Request and response parameters MUST NOT be included more than once.

3.2.1. Client Authentication

TOC

Confidential clients or other clients issued client credentials MUST authenticate with the authorization server as described in **Section 2.3** when making requests to the token endpoint. Client authentication is used for:

- Enforcing the binding of refresh tokens and authorization codes to the client they were issued to. Client authentication is critical when an authorization code is transmitted to the redirection endpoint over an insecure channel, or when the redirection URI has not been registered in full.
- Recovering from a compromised client by disabling the client or changing its credentials, thus preventing an attacker from abusing stolen refresh tokens. Changing a single set of client credentials is significantly faster than revoking an entire set of refresh tokens.
- Implementing authentication management best practices which require periodic credential rotation. Rotation of an entire set of refresh tokens can be challenging, while rotation of a single set of client credentials is significantly easier.

A public client that was not issued a client password MAY use the `client_id` request parameter to identify itself when sending requests to the token endpoint.

3.3. Access Token Scope

TOC

The authorization and token endpoints allow the client to specify the scope of the access request using the `scope` request parameter. In turn, the authorization server uses the `scope` response parameter to inform the client of the scope of the access token issued.

The value of the scope parameter is expressed as a list of space-delimited, case sensitive strings. The strings are 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.

```
scope       = scope-token *( SP scope-token )
scope-token = 1*( %x21 / %x23-5B / %x5D-7E )
```

The authorization server MAY fully or partially ignore the scope requested by the client based on the authorization server policy or the resource owner's instructions. If the issued access token scope is different from the one requested by the client, the authorization server MUST include the `scope` response parameter to inform the client of the actual scope granted.

If the client omits the scope parameter when requesting authorization, the authorization server MUST either process the request using a pre-defined default value, or fail the request indicating an invalid scope. The authorization server SHOULD document its scope requirements and default value (if defined).

---

4. Obtaining Authorization

TOC

To request an access token, the client obtains authorization from the resource owner. The authorization is expressed in the form of an authorization grant which the client uses to request the access token. OAuth defines four grant types: authorization code, implicit, resource owner password credentials, and client credentials. It also provides an extension mechanism for defining additional grant types.

4.1. Authorization Code Grant

TOC

The authorization code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients. As a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.

```
+----------+
| resource |
|   owner  |
|          |
+----------+
     ^
     |
    (B)
+----|-----+          Client Identifier      +---------------+
|         -+----(A)-- & Redirection URI ---->|               |
|  User-   |                                 | Authorization |
|  Agent  -+----(B)-- User authenticates --->|     Server    |
|          |                                 |               |
|         -+----(C)-- Authorization Code ---<|               |
+-|----|---+                                 +---------------+
  |    |                                         ^      v
 (A)  (C)                                        |      |
  |    |                                         |      |
  ^    v                                         |      |
+---------+                                      |      |
|         |>---(D)-- Authorization Code ---------'      |
|  Client |          & Redirection URI                 |
|         |                                            |
|         |<---(E)----- Access Token -------------------'
+---------+       (w/ Optional Refresh Token)
```

Note: The lines illustrating steps A, B, and C are broken into two parts as they pass through the user-agent.

Figure 3: Authorization Code Flow

The flow illustrated in **Figure 3** includes the following steps:

(A)
: The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

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

(C)
: Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorization code and any local state provided by the client earlier.

(D)
: The client requests an access token from the authorization server's token endpoint by including the authorization code received in the previous step. When making the request, the client authenticates with the authorization server. The client includes the redirection URI used to obtain the authorization code for verification.

(E)
: The authorization server authenticates the client, validates the authorization code, and ensures the redirection URI received matches the URI used to redirect the client in step (C). If valid, the authorization server responds back with an access token and optionally, a refresh token.

4.1.1. Authorization Request

TOC

The client constructs the request URI by adding the following parameters to the query component of the authorization endpoint URI using the `application/x-www-form-urlencoded` format as defined by **[W3C.REC-html401-19991224]**:

response_type
: REQUIRED. Value MUST be set to `code`.

client_id
: REQUIRED. The client identifier as described in **Section 2.2**.

redirect_uri
: OPTIONAL. As described in **Section 3.1.2**.

scope
: OPTIONAL. The scope of the access request as described by **Section 3.3**.

state
: RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client. The parameter SHOULD be used for preventing cross-site request forgery as described in **Section 10.12**.

The client directs the resource owner to the constructed URI using an HTTP redirection response, or by other means available to it via the user-agent.

For example, the client directs the user-agent to make the following HTTP request using TLS (extra line breaks are for display purposes only):

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

The authorization server validates the request to ensure all required parameters are present and valid. If the request is valid, the authorization server authenticates the resource owner and obtains an authorization decision (by asking the resource owner or by establishing approval via other means).

When a decision is established, the authorization server directs the user-agent to the provided client redirection URI using an HTTP redirection response, or by other means available to it via the user-agent.

TOC

4.1.2. Authorization Response

If the resource owner grants the access request, the authorization server issues an authorization code and delivers it to the client by adding the following parameters to the query component of the redirection URI using the `application/x-www-form-urlencoded` format:

code
: REQUIRED. The authorization code generated by the authorization server. The authorization code MUST expire shortly after it is issued to mitigate the risk of leaks. A maximum authorization code lifetime of 10 minutes is RECOMMENDED. The client MUST NOT use the authorization code more than once. If an authorization code is used more than once, the authorization server MUST deny the request and SHOULD revoke (when possible) all tokens previously issued based on that authorization code. The authorization code is bound to the client identifier and redirection URI.

state
: REQUIRED if the `state` parameter was present in the client authorization request. The exact value received from the client.

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

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA
          &state=xyz
```

The client MUST ignore unrecognized response parameters. The authorization code string size is left undefined by this specification. The client should avoid making assumptions about code value sizes. The authorization server should document the size of any value it issues.

4.1.2.1. Error Response

TOC

If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier is missing or invalid, the authorization server SHOULD inform the resource owner of the error, and MUST NOT automatically redirect the user-agent to the invalid redirection URI.

If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding the following parameters to the query component of the redirection URI using the `application/x-www-form-urlencoded` format:

error
: REQUIRED. A single error code from the following:

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

  unauthorized_client
  : The client is not authorized to request an authorization code using this method.

  access_denied
  : The resource owner or authorization server denied the request.

  unsupported_response_type
  : The authorization server does not support obtaining an authorization code using this method.

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

  server_error
  : The authorization server encountered an unexpected condition which prevented it from fulfilling the request.

  temporarily_unavailable
  : The authorization server is currently unable to handle the request due to a temporary overloading or maintenance of the server.

error_description
: OPTIONAL. A human-readable UTF-8 encoded text providing additional information, used to assist the client developer in understanding the error that occurred.

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

state
: REQUIRED if a `state` parameter was present in the client authorization request. The exact value received from the client.

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

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

4.1.3. Access Token Request

The client makes a request to the token endpoint by adding the following parameters using the `application/x-www-form-urlencoded` format in the HTTP request entity-body:

grant_type
: REQUIRED. Value MUST be set to `authorization_code`.

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

redirect_uri
: REQUIRED, if the `redirect_uri` parameter was included in the authorization request as described in **Section 4.1.1**, and their values MUST be identical.

If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in **Section 3.2.1**.

For example, the client makes the following HTTP request using TLS (extra 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;charset=UTF-8

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

The authorization server MUST:

- require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),
- authenticate the client if client authentication is included and ensure the authorization code was issued to the authenticated client,
- verify that the authorization code is valid, and
- ensure that the `redirect_uri` parameter is present if the `redirect_uri` parameter was included in the initial authorization request as described in **Section 4.1.1**, and if included ensure their values are identical.

4.1.4. Access Token Response

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in **Section 5.1**. If the request client authentication failed or is invalid, the authorization server returns an error response as described in **Section 5.2**.

An example successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
```

```
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter":"example_value"
}
```

TOC

4.2. Implicit Grant

The implicit grant type is used to obtain access tokens (it does not support the issuance of refresh tokens) and is optimized for public clients known to operate a particular redirection URI. These clients are typically implemented in a browser using a scripting language such as JavaScript.

As a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.

Unlike the authorization code grant type in which the client makes separate requests for authorization and access token, the client receives the access token as the result of the authorization request.

The implicit grant type does not include client authentication, and relies on the presence of the resource owner and the registration of the redirection URI. Because the access token is encoded into the redirection URI, it may be exposed to the resource owner and other applications residing on the same device.

```
+----------+
| Resource |
|  Owner   |
|          |
+----------+
     ^
     |
    (B)
+----|-----+          Client Identifier     +---------------+
|         -+----(A)-- & Redirection URI --->|               |
|  User-   |                                | Authorization |
|  Agent  -|----(B)-- User authenticates -->|     Server    |
|          |                                |               |
|          |<---(C)--- Redirection URI ----<|               |
|          |          with Access Token     +---------------+
|          |            in Fragment
|          |                                +---------------+
|          |----(D)--- Redirection URI ---->|   Web-Hosted  |
|          |          without Fragment      |     Client    |
|          |                                |    Resource   |
|     (F)  |<---(E)------- Script ---------<|               |
|          |                                +---------------+
+-|--------+
  |    |
 (A)  (G) Access Token
  |    |
  ^    v
+---------+
|         |
|  Client |
|         |
+---------+
```

Note: The lines illustrating steps A and B are broken into two parts as they pass through the user-agent.

Figure 4: Implicit Grant Flow

The flow illustrated in **Figure 4** includes the following steps:

(A)
: The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

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

(C)
: Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier. The redirection URI includes the access token in the URI fragment.

(D)
: The user-agent follows the redirection instructions by making a request to the web-hosted client resource (which does not include the fragment per **[RFC2616]**). The user-agent retains the fragment information locally.

(E)
: The web-hosted client resource returns a web page (typically an HTML document 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-hosted client resource locally, which extracts the access token and passes it to the client.

TOC

4.2.1. Authorization Request

The client constructs the request URI by adding the following parameters to the query component of the authorization endpoint URI using the `application/x-www-form-urlencoded` format:

response_type
: REQUIRED. Value MUST be set to `token`.

client_id
: REQUIRED. The client identifier as described in **Section 2.2**.

redirect_uri
: OPTIONAL. As described in **Section 3.1.2**.

scope
: OPTIONAL. The scope of the access request as described by **Section 3.3**.

state
: RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client. The parameter SHOULD be used for preventing cross-site request forgery as described in **Section 10.12**.

The client directs the resource owner to the constructed URI using an HTTP redirection response, or by other means available to it via the user-agent.

For example, the client directs the user-agent to make the following HTTP request using TLS (extra line breaks are for display purposes only):

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

The authorization server validates the request to ensure all required parameters are present and valid. The authorization server MUST verify that the redirection URI to which it will redirect the access token matches a redirection URI registered by the client as described in **Section 3.1.2**.

If the request is valid, the authorization server authenticates the resource owner and obtains an authorization decision (by asking the resource owner or by establishing approval via other means).

When a decision is established, the authorization server directs the user-agent to the provided client redirection URI using an HTTP redirection response, or by other means available to it via the user-agent.

4.2.2. Access Token Response

TOC

If the resource owner grants the access request, the authorization server issues an access token and delivers it to the client by adding the following parameters to the fragment component of the redirection URI using the `application/x-www-form-urlencoded` format:

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

token_type
: REQUIRED. The type of the token issued as described in **Section 7.1**. Value is case insensitive.

expires_in
: RECOMMENDED. The lifetime in seconds of the access token. For example, the value `3600` denotes that the access token will expire in one hour from the time the response was generated. If omitted, the authorization server SHOULD provide the expiration time via other means or document the default value.

scope
: OPTIONAL, if identical to the scope requested by the client, otherwise REQUIRED. The scope of the access token as described by **Section 3.3**.

state
: REQUIRED if the `state` parameter was present in the client authorization request. The exact value received from the client.

The authorization server MUST NOT issue a refresh token.

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

```
HTTP/1.1 302 Found
Location: http://example.com/cb#access_token=2YotnFZFEjr1zCsicMWpAA
          &state=xyz&token_type=example&expires_in=3600
```

Developers should note that some user-agents do not support the inclusion of a fragment component in the HTTP `Location` response header field. Such clients will require using other methods for redirecting the client than a 3xx redirection response. For example, returning an HTML page which includes a 'continue' button with an action linked to the redirection URI.

The client MUST ignore unrecognized response parameters. The access token string size is left undefined by this specification. The client should avoid making assumptions about value sizes. The authorization server should document the size of any value it issues.

4.2.2.1. Error Response

TOC

If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client

identifier is missing or invalid, the authorization server SHOULD inform the resource owner of the error, and MUST NOT automatically redirect the user-agent to the invalid redirection URI.

If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding the following parameters to the fragment component of the redirection URI using the `application/x-www-form-urlencoded` format:

error
: REQUIRED. A single error code from the following:

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

  unauthorized_client
  : The client is not authorized to request an access token using this method.

  access_denied
  : The resource owner or authorization server denied the request.

  unsupported_response_type
  : The authorization server does not support obtaining an access token using this method.

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

  server_error
  : The authorization server encountered an unexpected condition which prevented it from fulfilling the request.

  temporarily_unavailable
  : The authorization server is currently unable to handle the request due to a temporary overloading or maintenance of the server.

error_description
: OPTIONAL. A human-readable UTF-8 encoded text providing additional information, used to assist the client developer in understanding the error that occurred.

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

state
: REQUIRED if a `state` parameter was present in the client authorization request. The exact value received from the client.

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

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb#error=access_denied&state=xyz
```

TOC

4.3. Resource Owner Password Credentials Grant

The resource owner password credentials grant type is suitable in cases where the resource owner has a trust relationship with the client, such as the device operating system or a highly privileged application. The authorization server should take special care when enabling this grant type, and only allow it when other flows are not viable.

The grant type is suitable for clients capable of obtaining the resource owner's credentials (username and password, typically using an interactive form). It is also used to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials to an access token.

```
+----------+
| Resource |
|  Owner   |
|          |
+----------+
     v
     |    Resource Owner
    (A) Password Credentials
     |
     v
+---------+                                  +---------------+
|         |>--(B)---- Resource Owner ------->|               |
|         |         Password Credentials     | Authorization |
| Client  |                                  |     Server    |
|         |<--(C)---- Access Token ---------<|               |
|         |    (w/ Optional Refresh Token)   |               |
+---------+                                  +---------------+
```

Figure 5: Resource Owner Password Credentials Flow

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

(A)
: The resource owner provides the client with its username and password.

(B)
: The client requests an access token from the authorization server's token endpoint by including the credentials received from the resource owner. When making the request, the client authenticates with the authorization server.

(C)
: The authorization server authenticates the client and validates the resource owner credentials, and if valid issues an access token.

4.3.1. Authorization Request and Response

TOC

The method through which the client obtains the resource owner credentials is beyond the scope of this specification. The client MUST discard the credentials once an access token has been obtained.

4.3.2. Access Token Request

TOC

The client makes a request to the token endpoint by adding the following parameters using the `application/x-www-form-urlencoded` format in the HTTP request entity-body:

grant_type
: REQUIRED. Value MUST be set to `password`.

username
: REQUIRED. The resource owner username, encoded as UTF-8.

password
: REQUIRED. The resource owner password, encoded as UTF-8.

scope
: OPTIONAL. The scope of the access request as described by **Section 3.3**.

If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in **Section 3.2.1**.

For example, the client makes the following HTTP request using transport-layer security (extra 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;charset=UTF-8

grant_type=password&username=johndoe&password=A3ddj3w
```

The authorization server MUST:

- require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),
- authenticate the client if client authentication is included, and
- validate the resource owner password credentials.

Since this access token request utilizes the resource owner's password, the authorization server MUST protect the endpoint against brute force attacks (e.g. using rate-limitation or generating alerts).

4.3.3. Access Token Response

TOC

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in **Section 5.1**. If the request failed client authentication or is invalid, the authorization server returns an error response as described in **Section 5.2**.

An example successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter":"example_value"
}
```

4.4. Client Credentials Grant

TOC

The client can request an access token using only its client credentials (or other supported means of authentication) when the client is requesting access to the protected resources under its control, or those of another resource owner which has been previously arranged with the authorization server (the method of which is beyond the scope of this specification).

The client credentials grant type MUST only be used by confidential clients.

```
+---------+                                  +---------------+
|         |                                  |               |
```

```
  |         |>--(A)- Client Authentication --->| Authorization |
  | Client  |                                  |     Server    |
  |         |<--(B)---- Access Token ---------<|               |
  |         |                                  |               |
  +---------+                                  +---------------+
```

Figure 6: Client Credentials Flow

The flow illustrated in **Figure 6** includes the following steps:

(A)
: The client authenticates with the authorization server and requests an access token from the token endpoint.

(B)
: The authorization server authenticates the client, and if valid issues an access token.

4.4.1. Authorization Request and Response

TOC

Since the client authentication is used as the authorization grant, no additional authorization request is needed.

4.4.2. Access Token Request

TOC

The client makes a request to the token endpoint by adding the following parameters using the `application/x-www-form-urlencoded` format in the HTTP request entity-body:

grant_type
: REQUIRED. Value MUST be set to `client_credentials`.

scope
: OPTIONAL. The scope of the access request as described by **Section 3.3**.

The client MUST authenticate with the authorization server as described in **Section 3.2.1**.

For example, the client makes the following HTTP request using transport-layer security (extra 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;charset=UTF-8

grant_type=client_credentials
```

The authorization server MUST authenticate the client.

4.4.3. Access Token Response

TOC

If the access token request is valid and authorized, the authorization server issues an access token as described in **Section 5.1**. A refresh token SHOULD NOT be included. If the request failed client authentication or is invalid, the authorization server returns an error response as described in **Section 5.2**.

An example successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "example_parameter":"example_value"
}
```

4.5. Extension Grants

TOC

The client uses an extension grant type by specifying the grant type using an absolute URI (defined by the authorization server) as the value of the `grant_type` parameter of the token endpoint, and by adding any additional parameters necessary.

For example, to request an access token using a SAML 2.0 assertion grant type as defined by **[I-D.ietf-oauth-saml2-bearer]**, the client makes the following HTTP request using TLS (line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded;charset=UTF-8

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Asaml2-
bearer&assertion=PEFzc2VydGlvbiBJc3N1ZUluc3RhbnQ9IjIwMTEtMDU
[...omitted for brevity...]aG5TdGF0ZW1lbnQ-PC9Bc3NlcnRpb24-
```

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in **Section 5.1**. If the request failed client authentication or is invalid, the authorization server returns an error response as described in **Section 5.2**.

---

5. Issuing an Access Token

TOC

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in **Section 5.1**. If the request failed client authentication or is invalid, the authorization server returns an error response as described in **Section 5.2**.

5.1. Successful Response

TOC

The authorization server issues an access token and optional refresh token, and constructs the response by adding the following parameters to the entity body of the HTTP response with a 200 (OK) status code:

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

token_type
: REQUIRED. The type of the token issued as described in **Section 7.1**. Value is

case insensitive.

expires_in
: RECOMMENDED. The lifetime in seconds of the access token. For example, the value `3600` denotes that the access token will expire in one hour from the time the response was generated. If omitted, the authorization server SHOULD provide the expiration time via other means or document the default value.

refresh_token
: OPTIONAL. The refresh token which can be used to obtain new access tokens using the same authorization grant as described in **Section 6**.

scope
: OPTIONAL, if identical to the scope requested by the client, otherwise REQUIRED. The scope of the access token as described by **Section 3.3**.

The parameters are included 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 order of parameters does not matter and can vary.

The authorization server MUST include the HTTP `Cache-Control` response header field **[RFC2616]** with a value of `no-store` in any response containing tokens, credentials, or other sensitive information, as well as the `Pragma` response header field **[RFC2616]** with a value of `no-cache`.

For example:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter":"example_value"
}
```

The client MUST ignore unrecognized value names in the response. The sizes of tokens and other values received from the authorization server are left undefined. The client should avoid making assumptions about value sizes. The authorization server should document the size of any value it issues.

TOC

5.2. Error Response

The authorization server responds with an HTTP 400 (Bad Request) status code and includes the following parameters with the response:

error
: REQUIRED. A single error code from the following:

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

  invalid_client
  : Client authentication failed (e.g. unknown client, no client authentication included, or unsupported authentication method). The authorization server MAY return an HTTP 401

(Unauthorized) status code to indicate which HTTP authentication schemes are supported. If the client attempted to authenticate via the `Authorization` request header field, the authorization server MUST respond with an HTTP 401 (Unauthorized) status code, and include the `WWW-Authenticate` response header field matching the authentication scheme used by the client.

invalid_grant
: The provided authorization grant (e.g. authorization code, resource owner credentials) is invalid, expired, revoked, does not match the redirection URI used in the authorization request, or was issued to another client.

unauthorized_client
: The authenticated client is not authorized to use this authorization grant type.

unsupported_grant_type
: The authorization grant type is not supported by the authorization server.

invalid_scope
: The requested scope is invalid, unknown, malformed, or exceeds the scope granted by the resource owner.

error_description
: OPTIONAL. A human-readable UTF-8 encoded text providing additional information, used to assist the client developer in understanding the error that occurred.

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

The parameters are included 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 order of parameters does not matter and can vary.

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

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

TOC

6. Refreshing an Access Token

If the authorization server issued a refresh token to the client, the client makes a refresh request to the token endpoint by adding the following parameters using the `application/x-www-form-urlencoded` format in the HTTP request entity-body:

grant_type
: REQUIRED. Value MUST be set to `refresh_token`.

refresh_token
: REQUIRED. The refresh token issued to the client.

scope
: OPTIONAL. The scope of the access request as described by **Section 3.3**. The requested scope MUST NOT include any scope not originally granted by the resource owner, and if omitted is treated as equal to the scope originally granted

by the resource owner.

Because refresh tokens are typically long-lasting credentials used to request additional access tokens, the refresh token is bound to the client which it was issued. If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in **Section 3.2.1**.

For example, the client makes the following HTTP request using transport-layer security (extra 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;charset=UTF-8

grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
```

The authorization server MUST:

- require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),
- authenticate the client if client authentication is included and ensure the refresh token was issued to the authenticated client, and
- validate the refresh token.

If valid and authorized, the authorization server issues an access token as described in **Section 5.1**. If the request failed verification or is invalid, the authorization server returns an error response as described in **Section 5.2**.

The authorization server MAY issue a new refresh token, in which case the client MUST discard the old refresh token and replace it with the new refresh token. The authorization server MAY revoke the old refresh token after issuing a new refresh token to the client. If a new refresh token is issued, the refresh token scope MUST be identical to that of the refresh token included by the client in the request.

---

7. Accessing Protected Resources

TOC

The client accesses protected resources by presenting the access token to the resource server. The resource server MUST validate the access token and ensure it has not expired and that its scope covers the requested resource. The methods used by the resource server to validate the access token (as well as any error responses) are beyond the scope of this specification, but generally involve an interaction or coordination between the resource server and the authorization server.

The method in which the client utilized the access token to authenticate with the resource server depends on the type of access token issued by the authorization server. Typically, it involves using the HTTP `Authorization` request header field **[RFC2617]** with an authentication scheme defined by the access token type specification.

7.1. Access Token Types

TOC

The access token type provides the client with the information required to successfully utilize the access token to make a protected resource request (along with type-specific attributes). The client MUST NOT use an access token if it does not understand the token type.

For example, the `bearer` token type defined in **[I-D.ietf-oauth-v2-bearer]** is utilized by simply including the access token string in the request:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: Bearer 7Fjfp0ZBr1KtDRbnfVdmIw
```

while the `mac` token type defined in **[I-D.ietf-oauth-v2-http-mac]** is utilized by issuing a MAC key together with the access token which is used to sign certain components of the HTTP requests:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: MAC id="h480djs93hd8",
                   nonce="274312:dj83hs9s",
                   mac="kDZvddkndxvhGRXZhvuDjEWhGeE="
```

The above examples are provided for illustration purposes only. Developers are advised to consult the **[I-D.ietf-oauth-v2-bearer]** and **[I-D.ietf-oauth-v2-http-mac]** specifications before use.

Each access token type definition specifies the additional attributes (if any) sent to the client together with the `access_token` response parameter. It also defines the HTTP authentication method used to include the access token when making a protected resource request.

8. Extensibility

TOC

8.1. Defining Access Token Types

TOC

Access token types can be defined in one of two ways: registered in the access token type registry (following the procedures in **Section 11.1**), or by using a unique absolute URI as its name.

Types utilizing a URI name SHOULD be limited to vendor-specific implementations that are not commonly applicable, and are specific to the implementation details of the resource server where they are used.

All other types MUST be registered. Type names MUST conform to the type-name ABNF. If the type definition includes a new HTTP authentication scheme, the type name SHOULD be identical to the HTTP authentication scheme name (as defined by **[RFC2617]**). The token type `example` is reserved for use in examples.

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

8.2. Defining New Endpoint Parameters

TOC

New request or response parameters for use with the authorization endpoint or the token endpoint are defined and registered in the parameters registry following the procedure in **Section 11.2**.

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

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

Unregistered vendor-specific parameter extensions that are not commonly applicable, and are specific to the implementation details of the authorization server where they are used SHOULD utilize a vendor-specific prefix that is not likely to conflict with other registered values (e.g. begin with 'companyname_').

8.3. Defining New Authorization Grant Types

TOC

New authorization grant types can be defined by assigning them a unique absolute URI for use with the `grant_type` parameter. If the extension grant type requires additional token endpoint parameters, they MUST be registered in the OAuth parameters registry as described by **Section 11.2**.

8.4. Defining New Authorization Endpoint Response Types

TOC

New response types for use with the authorization endpoint are defined and registered in the authorization endpoint response type registry following the procedure in **Section 11.3**. Response type names MUST conform to the response-type ABNF.

```
response-type  = response-name *( SP response-name )
response-name  = 1*response-char
response-char  = "_" / DIGIT / ALPHA
```

If a response type contains one or more space characters (%x20), it is compared as a space-delimited list of values in which the order of values does not matter. Only one order of values can be registered, which covers all other arrangements of the same set of values.

For example, the response type `token code` is left undefined by this specification. However, an extension can define and register the `token code` response type. Once registered, the same combination cannot be registered as `code token`, but both values can be used to denote the same response type.

8.5. Defining Additional Error Codes

TOC

In cases where protocol extensions (i.e. access token types, extension parameters, or extension grant types) require additional error codes to be used with the authorization code grant error response (**Section 4.1.2.1**), the implicit grant error response (**Section 4.2.2.1**), or the token error response (**Section 5.2**), such error codes MAY be defined.

Extension error codes MUST be registered (following the procedures in **Section 11.4**) if the extension they are used in conjunction with is a registered access token type, a registered endpoint parameter, or an extension grant type. Error codes used with unregistered extensions MAY be registered.

Error codes MUST conform to the error-code ABNF, and SHOULD be prefixed by an

identifying name when possible. For example, an error identifying an invalid value set to the extension parameter `example` should be named `example_invalid`.

```
error-code   = ALPHA *error-char
error-char   = "-" / "." / "_" / DIGIT / ALPHA
```

---

9. Native Applications

TOC

Native applications are clients installed and executed on the device used by the resource owner (i.e. desktop application, native mobile application). Native applications require special consideration related to security, platform capabilities, and overall end-user experience.

The authorization endpoint requires interaction between the client and the resource owner's user-agent. Native applications can invoke an external user-agent or embed a user-agent within the application. For example:

- External user-agent - the native application can capture the response from the authorization server using a redirection URI with a scheme registered with the operating system to invoke the client as the handler, manual copy-and-paste of the credentials, running a local web server, installing a user-agent extension, or by providing a redirection URI identifying a server-hosted resource under the client's control, which in turn makes the response available to the native application.
- Embedded user-agent - the native application obtains the response by directly communicating with the embedded user-agent by monitoring state changes emitted during the resource load, or accessing the user-agent's cookies storage.

When choosing between an external or embedded user-agent, developers should consider:

- External user-agents may improve completion rate as the resource owner may already have an active session with the authorization server removing the need to re-authenticate. It provides a familiar end-user experience and functionality. The resource owner may also rely on user-agent features or extensions to assist with authentication (e.g. password manager, 2-factor device reader).
- Embedded user-agents may offer improved usability, as they remove the need to switch context and open new windows.
- Embedded user-agents pose a security challenge because resource owners are authenticating in an unidentified window without access to the visual protections found in most external user-agents. Embedded user-agents educate end-user to trust unidentified requests for authentication (making phishing attacks easier to execute).

When choosing between the implicit grant type and the authorization code grant type, the following should be considered:

- Native applications that use the authorization code grant type SHOULD do so without using client credentials, due to the native application's inability to keep client credentials confidential.
- When using the implicit grant type flow a refresh token is not returned which requires repeating the authorization process once the access token expires.

---

10. Security Considerations

TOC

As a flexible and extensible framework, OAuth's security considerations depend on many factors. The following sections provide implementers with security guidelines focused on the three client profiles described in **Section 2.1**: web application, user-agent-based application, and native application.

A comprehensive OAuth security model and analysis, as well as background for the protocol design is provided by **[I-D.ietf-oauth-v2-threatmodel]**.

10.1. Client Authentication

TOC

The authorization server establishes client credentials with web application clients for the purpose of client authentication. The authorization server is encouraged to consider stronger client authentication means than a client password. Web application clients MUST ensure confidentiality of client passwords and other client credentials.

The authorization server MUST NOT issue client passwords or other client credentials to native application or user-agent-based application clients for the purpose of client authentication. The authorization server MAY issue a client password or other credentials for a specific installation of a native application client on a specific device.

When client authentication is not possible, the authorization server SHOULD employ other means to validate the client's identity. For example, by requiring the registration of the client redirection URI or enlisting the resource owner to confirm identity. A valid redirection URI is not sufficient to verify the client's identity when asking for end-user authorization, but can be used to prevent delivering credentials to a counterfeit client after obtaining end-user authorization.

The authorization server must consider the security implications of interacting with unauthenticated clients and take measures to limit the potential exposure of other credentials (e.g. refresh tokens) issued to such clients.

10.2. Client Impersonation

TOC

A malicious client can impersonate another client and obtain access to protected resources, if the impersonated client fails to, or is unable to, keep its client credentials confidential.

The authorization server MUST authenticate the client whenever possible. If the authorization server cannot authenticate the client due to the client's nature, the authorization server MUST require the registration of any redirection URI used for receiving authorization responses, and SHOULD utilize other means to protect resource owners from such potentially malicious clients. For example, the authorization server can engage the resource owner to assist in identifying the client and its origin.

The authorization server SHOULD enforce explicit resource owner authentication and provide the resource owner with information about the client and the requested authorization scope and lifetime. It is up to the resource owner to review the information in the context of the current client, and authorize or deny the request.

The authorization server SHOULD NOT process repeated authorization requests automatically (without active resource owner interaction) without authenticating the client or relying on other measures to ensure the repeated request comes from the original client and not an impersonator.

10.3. Access Tokens

TOC

Access token credentials (as well as any confidential access token attributes) MUST be kept confidential in transit and storage, and only shared among the authorization server, the resource servers the access token is valid for, and the client to whom the access token is issued. Access token credentials MUST only be transmitted using TLS as described in **Section 1.6** with server authentication as defined by **[RFC2818]**.

When using the implicit grant type, the access token is transmitted in the URI fragment, which can expose it to unauthorized parties.

The authorization server MUST ensure that access tokens cannot be generated, modified, or guessed to produce valid access tokens by unauthorized parties.

The client SHOULD request access tokens with the minimal scope and lifetime necessary.

The authorization server SHOULD take the client identity into account when choosing how to honor the requested scope and lifetime, and MAY issue an access token with a less rights than requested.

10.4. Refresh Tokens

TOC

Authorization servers MAY issue refresh tokens to web application clients and native application clients.

Refresh tokens MUST be kept confidential in transit and storage, and shared only among the authorization server and the client to whom the refresh tokens were issued. The authorization server MUST maintain the binding between a refresh token and the client to whom it was issued. Refresh tokens MUST only be transmitted using TLS as described in **Section 1.6** with server authentication as defined by **[RFC2818]**.

The authorization server MUST verify the binding between the refresh token and client identity whenever the client identity can be authenticated. When client authentication is not possible, the authorization server SHOULD deploy other means to detect refresh token abuse.

For example, the authorization server could employ refresh token rotation in which a new refresh token is issued with every access token refresh response. The previous refresh token is invalidated but retained by the authorization server. If a refresh token is compromised and subsequently used by both the attacker and the legitimate client, one of them will present an invalidated refresh token which will inform the authorization server of the breach.

The authorization server MUST ensure that refresh tokens cannot be generated, modified, or guessed to produce valid refresh tokens by unauthorized parties.

10.5. Authorization Codes

TOC

The transmission of authorization codes SHOULD be made over a secure channel, and the client SHOULD implement TLS for use with its redirection URI if the URI identifies a network resource. Since authorization codes are transmitted via user-agent redirections, they could potentially be disclosed through user-agent history and HTTP referrer headers.

Authorization codes operate as plaintext bearer credentials, used to verify that the resource owner who granted authorization at the authorization server is the same resource owner returning to the client to complete the process. Therefore, if the client relies on the authorization code for its own resource owner authentication, the client redirection endpoint MUST require TLS.

Authorization codes MUST be short lived and single use. If the authorization server observes multiple attempts to exchange an authorization code for an access token, the authorization server SHOULD attempt to revoke all access tokens already granted based on the compromised authorization code.

If the client can be authenticated, the authorization servers MUST authenticate the client and ensure that the authorization code was issued to the same client.

10.6. Authorization Code Redirection URI Manipulation

TOC

When requesting authorization using the authorization code grant type, the client can specify a redirection URI via the `redirect_uri` parameter. If an attacker can manipulate the value of the redirection URI, it can cause the authorization server to redirect the resource owner user-agent to a URI under the control of the attacker with the authorization code.

An attacker can create an account at a legitimate client and initiate the authorization flow. When the attacker is sent to the authorization server to grant access, the attacker grabs the authorization URI provided by the legitimate client, and replaces the client's redirection URI with a URI under the control of the attacker. The attacker then tricks the victim into following

the manipulated link to authorize access to the legitimate client.

Once at the authorization server, the victim is prompted with a normal, valid request on behalf of a legitimate and trusted client, and authorizes the request. The victim is then redirected to an endpoint under the control of the attacker with the authorization code. The attacker completes the authorization flow by sending the authorization code to the client using the original redirection URI provided by the client. The client exchanges the authorization code with an access token and links it to the attacker's client account which can now gain access to the protected resources authorized by the victim (via the client).

In order to prevent such an attack, the authorization server MUST ensure that the redirection URI used to obtain the authorization code is identical to the redirection URI provided when exchanging the authorization code for an access token. The authorization server MUST require public clients and SHOULD require confidential clients to register their redirection URIs. If a redirection URI is provided in the request, the authorization server MUST validate it against the registered value.

10.7. Resource Owner Password Credentials

TOC

The resource owner password credentials grant type is often used for legacy or migration reasons. It reduces the overall risk of storing username and password by the client, but does not eliminate the need to expose highly privileged credentials to the client.

This grant type carries a higher risk than other grant types because it maintains the password anti-pattern this protocol seeks to avoid. The client could abuse the password or the password could unintentionally be disclosed to an attacker (e.g. via log files or other records kept by the client).

Additionally, because the resource owner does not have control over the authorization process (the resource owner involvement ends when it hands over its credentials to the client), the client can obtain access tokens with a broader scope and longer lifetime than desired by the resource owner. The authorization server should consider the scope and lifetime of access tokens issued via this grant type.

The authorization server and client SHOULD minimize use of this grant type and utilize other grant types whenever possible.

10.8. Request Confidentiality

TOC

Access tokens, refresh tokens, resource owner passwords, and client credentials MUST NOT be transmitted in the clear. Authorization codes SHOULD NOT be transmitted in the clear.

The `state` and `scope` parameters SHOULD NOT include sensitive client or resource owner information in plain text as they can be transmitted over insecure channels or stored insecurely.

10.9. Endpoints Authenticity

TOC

In order to prevent man-in-the-middle attacks, the authorization server MUST implement and require TLS with server authentication as defined by **[RFC2818]** for any request sent to the authorization and token endpoints. The client MUST validate the authorization server's TLS certificate in accordance with its requirements for server identity authentication.

10.10. Credentials Guessing Attacks

TOC

The authorization server MUST prevent attackers from guessing access tokens, authorization codes, refresh tokens, resource owner passwords, and client credentials.

Generated tokens and other credentials not intended for handling by end-users MUST be constructed from a cryptographically strong random or pseudo-random number sequence (**[RFC1750]**) generated by the authorization server. The probability of any two values being identical MUST be less than or equal to 2^(-128) and SHOULD be less than or equal to 2^(-160).

The authorization server MUST utilize other means to protect credentials intended for end-user usage.

10.11. Phishing Attacks

TOC

Wide deployment of this and similar protocols may cause end-users to become inured to the practice of being redirected to websites where they are asked to enter their passwords. If end-users are not careful to verify the authenticity of these websites before entering their credentials, it will be possible for attackers to exploit this practice to steal resource owners' passwords.

Service providers should attempt to educate end-users about the risks phishing attacks pose, and should provide mechanisms that make it easy for end-users to confirm the authenticity of their sites. Client developers should consider the security implications of how they interact with the user-agent (e.g., external, embedded), and the ability of the end-user to verify the authenticity of the authorization server.

To reduce the risk of phishing attacks, the authorization servers MUST utilize TLS on every endpoint used for end-user interaction.

10.12. Cross-Site Request Forgery

TOC

Cross-site request forgery (CSRF) is an exploit in which an attacker causes the user-agent of a victim end-user to follow a malicious URI (e.g. provided to the user-agent as a misleading link, image, or redirection) to a trusting server (usually established via the presence of a valid session cookie).

A CSRF attack against the client's redirection URI allows an attacker to inject their own authorization code or access token, which can result in the client using an access token associated with the attacker's protected resources rather than the victim's (e.g. save the victim's bank account information to a protected resource controlled by the attacker).

The client MUST implement CSRF protection for its redirection URI. This is typically accomplished by requiring any request sent to the redirection URI endpoint to include a value that binds the request to the user-agent's authenticated state (e.g. a hash of the session cookie used to authenticate the user-agent). The client SHOULD utilize the `state` request parameter to deliver this value to the authorization server when making an authorization request.

Once authorization has been obtained from the end-user, the authorization server redirects the end-user's user-agent back to the client with the required binding value contained in the `state` parameter. The binding value enables the client to verify the validity of the request by matching the binding value to the user-agent's authenticated state. The binding value used for CSRF protection MUST contain a non-guessable value (as described in **Section 10.10**), and the user-agent's authenticated state (e.g. session cookie, HTML5 local storage) MUST be kept in a location accessible only to the client and the user-agent (i.e., protected by same-origin policy).

A CSRF attack against the authorization server's authorization endpoint can result in an attacker obtaining end-user authorization for a malicious client without involving or alerting the end-user.

The authorization server MUST implement CSRF protection for its authorization endpoint, and ensure that a malicious client cannot obtain authorization without the awareness and explicit consent of the resource owner.

10.13. Clickjacking

TOC

In a clickjacking attack, an attacker registers a legitimate client and then constructs a malicious site in which it loads the authorization server's authorization endpoint web page in a transparent iframe overlaid on top of a set of dummy buttons which are carefully constructed to be placed directly under important buttons on the authorization page. When an end-user clicks a misleading visible button, the end-user is actually clicking an invisible button on the authorization page (such as an "Authorize" button). This allows an attacker to trick a resource owner into granting its client access without their knowledge.

To prevent this form of attack, native applications SHOULD use external browsers instead of embedding browsers within the application when requesting end-user authorization. For most newer browsers, avoidance of iframes can be enforced by the authorization server using the (non-standard) `x-frame-options` header. This header can have two values, `deny` and `sameorigin`, which will block any framing, or framing by sites with a different origin, respectively. For older browsers, javascript framebusting techniques can be used but may not be effective in all browsers.

10.14. Code Injection and Input Validation

TOC

A code injection attack occurs when an input or otherwise external variable is used by an application unsanitized and causes modification to the application logic. This may allow an attacker to gain access to the application device or its data, cause denial of service, or a wide range of malicious side-effects.

The Authorization server and client MUST validate and sanitize any value received, and in particular, the value of the `state` and `redirect_uri` parameters.

10.15. Open Redirectors

TOC

The authorization server authorization endpoint and the client redirection endpoint can be improperly configured and operate as open redirectors. An open redirector is an endpoint using a parameter to automatically redirect a user-agent to the location specified by the parameter value without any validation.

Open redirectors can be used in phishing attacks, or by an attacker to get end-users to visit malicious sites by making the URI's authority look like a familiar and trusted destination. In addition, if the authorization server allows the client to register only part of the redirection URI, an attacker can use an open redirector operated by the client to construct a redirection URI that will pass the authorization server validation but will send the authorization code or access token to an endpoint under the control of the attacker.

---

11. IANA Considerations

TOC

11.1. The OAuth Access Token Type Registry

TOC

This specification establishes the OAuth access token type registry.

Access token types are registered on the advice of one or more Designated Experts (appointed by the IESG or their delegate), with a Specification Required (using terminology from **[RFC5226]**). However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests should be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for access token type: example"). [[

Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: oauth-ext-review.]]

Within at most 14 days of the request, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

Decisions (or lack thereof) made by the Designated Expert(s) can be first appealed to Security Area Directors (contactable using app-ads@tools.ietf.org email address or directly by looking up their email addresses on http://www.iesg.org/ website) and, if the appellant is not satisfied with the response, to the full IESG (using the iesg@iesg.org mailing list).

IANA should only accept registry updates from the Designated Expert(s), and should direct all requests for registration to the review mailing list.

11.1.1. Registration Template

TOC

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

Additional Token Endpoint Response Parameters:
: Additional response parameters returned together with the `access_token` parameter. New parameters MUST be separately registered in the OAuth parameters registry as described by **Section 11.2**.

HTTP Authentication Scheme(s):
: The HTTP authentication scheme name(s), if any, used to authenticate protected resources requests using access tokens of this type.

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

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

11.2. The OAuth Parameters Registry

TOC

This specification establishes the OAuth parameters registry.

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

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

Within at most 14 days of the request, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

Decisions (or lack thereof) made by the Designated Expert(s) can be first appealed to Security Area Directors (contactable using app-ads@tools.ietf.org email address or directly by looking up their email addresses on http://www.iesg.org/ website) and, if the appellant is not satisfied with the response, to the full IESG (using the iesg@iesg.org mailing list).

IANA should only accept registry updates from the Designated Expert(s), and should direct all

requests for registration to the review mailing list.

11.2.1. Registration Template

TOC

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

Parameter usage location:
: The location(s) where parameter can be used. The possible locations are: authorization request, authorization response, token request, or token response.

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

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

11.2.2. Initial Registry Contents

TOC

The OAuth Parameters Registry's initial contents are:

- Parameter name: client_id
- Parameter usage location: authorization request, token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: client_secret
- Parameter usage location: token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: response_type
- Parameter usage location: authorization request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: redirect_uri
- Parameter usage location: authorization request, token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: scope
- Parameter usage location: authorization request, authorization response, token request, token response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: state
- Parameter usage location: authorization request, authorization response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: code
- Parameter usage location: authorization response, token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: error_description
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: error_uri

- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: grant_type
- Parameter usage location: token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: access_token
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: token_type
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: expires_in
- Parameter usage location: authorization response, token response
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: username
- Parameter usage location: token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: password
- Parameter usage location: token request
- Change controller: IETF
- Specification document(s): [[this document]]

- Parameter name: refresh_token
- Parameter usage location: token request, token response
- Change controller: IETF
- Specification document(s): [[this document]]

11.3. The OAuth Authorization Endpoint Response Type Registry

TOC

This specification establishes the OAuth authorization endpoint response type registry.

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

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

Within at most 14 days of the request, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

Decisions (or lack thereof) made by the Designated Expert(s) can be first appealed to Security Area Directors (contactable using app-ads@tools.ietf.org email address or directly by looking up their email addresses on http://www.iesg.org/ website) and, if the appellant is not satisfied with the response, to the full IESG (using the iesg@iesg.org mailing list).

IANA should only accept registry updates from the Designated Expert(s), and should direct all requests for registration to the review mailing list.

11.3.1. Registration Template

TOC

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

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

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

11.3.2. Initial Registry Contents

TOC

The OAuth Authorization Endpoint Response Type Registry's initial contents are:

- Response type name: code
- Change controller: IETF
- Specification document(s): [[this document]]

- Response type name: token
- Change controller: IETF
- Specification document(s): [[this document]]

11.4. The OAuth Extensions Error Registry

TOC

This specification establishes the OAuth extensions error registry.

Additional error codes used together with other protocol extensions (i.e. extension grant types, access token types, or extension parameters) are registered on the advice of one or more Designated Experts (appointed by the IESG or their delegate), with a Specification Required (using terminology from **[RFC5226]**). However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

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

Within at most 14 days of the request, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

Decisions (or lack thereof) made by the Designated Expert(s) can be first appealed to Security Area Directors (contactable using app-ads@tools.ietf.org email address or directly by looking up their email addresses on http://www.iesg.org/ website) and, if the appellant is not satisfied with the response, to the full IESG (using the iesg@iesg.org mailing list).

IANA should only accept registry updates from the Designated Expert(s), and should direct all requests for registration to the review mailing list.

11.4.1. Registration Template

TOC

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

Error usage location:
: The location(s) where the error can be used. The possible locations are: authorization code grant error response (**Section 4.1.2.1**), implicit grant error response (**Section 4.2.2.1**), or token error response (**Section 5.2**).

Related protocol extension:
: The name of the extension grant type, access token type, or extension parameter, the error code is used in conjunction with.

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

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

---

12. Acknowledgements

TOC

The initial OAuth 2.0 protocol specification was edited by David Recordon, based on two previous publications: the OAuth 1.0 community specification **[RFC5849]**, and OAuth WRAP (OAuth Web Resource Authorization Profiles) **[I-D.draft-hardt-oauth-01]**. The Security Considerations section was drafted by Torsten Lodderstedt, Mark McGloin, Phil Hunt, and Anthony Nadalin.

The OAuth 1.0 community specification was edited by Eran Hammer and authored by Mark Atwood, Dirk Balfanz, Darren Bounds, Richard M. Conlan, Blaine Cook, Leah Culver, Breno de Medeiros, Brian Eaton, Kellan Elliott-McCrea, Larry Halff, Eran Hammer, Ben Laurie, Chris Messina, John Panzer, Sam Quigley, David Recordon, Eran Sandler, Jonathan Sergent, Todd Sieling, Brian Slesinsky, and Andy Smith.

The OAuth WRAP specification was edited by Dick Hardt and authored by Brian Eaton, Yaron Goland, Dick Hardt, and Allen Tom.

This specification is the work of the OAuth Working Group which includes dozens of active and dedicated participants. In particular, the following individuals contributed ideas, feedback, and wording which shaped and formed the final specification:

Michael Adams, Amanda Anganes, Andrew Arnott, Dirk Balfanz, Aiden Bell, Brian Campbell, Scott Cantor, Marcos Caceres, Blaine Cook, Roger Crew, Brian Eaton, Leah Culver, Bill de h ra, Andr DeMarre, Brian Eaton, Wolter Eldering, Brian Ellin, Igor Faynberg, George Fletcher, Tim Freeman, Evan Gilbert, Yaron Goland, Brent Goldman, Kristoffer Gronowski, Justin Hart, Dick Hardt, Craig Heath, Phil Hunt, Michael B. Jones, Terry Jones, John Kemp, Mark Kent, Raffi Krikorian, Chasen Le Hara, Rasmus Lerdorf, Torsten Lodderstedt, Hui-Lan Lu, Casey Lucas, Paul Madsen, Alastair Mair, Eve Maler, James Manger, Mark McGloin, Laurence Miao, William Mills, Chuck Mortimore, Anthony Nadalin, Julian Reschke, Justin Richer, Peter Saint-Andre, Nat Sakimura, Rob Sayre, Marius Scurtescu, Naitik Shah, Luke Shepard, Vlad Skvortsov, Justin Smith, Niv Steingarten, Christian St bner, Jeremy Suriel, Paul Tarjan, Allen Tom, Franklin Tse, Nick Walker, Shane Weeden, and Skylar Woodward.

This document was produced under the chairmanship of Blaine Cook, Peter Saint-Andre, Hannes Tschofenig, and Barry Leiba. The area directors included Lisa Dusseault, Peter Saint-Andre, and Stephen Farrell.

Appendix A. Editor's Notes

TOC

While many people contributed to this specification throughout its long journey, the editor would like to acknowledge and thank a few individuals for their outstanding and invaluable efforts leading up to the publication of this specification. It is these individuals without whom this work would not have existed or reached its successful conclusion.

David Recordon for continuously being one of OAuth s most valuable assets, bringing pragmatism and urgency to the work, and helping shape it from its very beginning, as well as being one of the best collaborators I had the pleasure of working with.

James Manger for his creative ideas and always insightful feedback. Brian Campbell, Torsten Lodderstedt, Chuck Mortimore, Justin Richer, Marius Scurtescu, and Luke Shepard for their continued participation and valuable feedback.

Special thanks goes to Mike Curtis and Yahoo! for their unconditional support of this work for over three years.

13. References

TOC

13.1. Normative References

TOC

[RFC1750] Eastlake, D., Crocker, S., and J. Schiller, "Randomness Recommendations for Security," RFC 1750, December 1994 (TXT).

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

[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0," RFC 2246, January 1999 (TXT).

[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).

[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).

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2," RFC 4949, August 2007 (TXT).

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

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF," STD 68, RFC 5234, January 2008 (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).

13.2. Informative References

TOC

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

[I-D.ietf-oauth-saml2-bearer] Mortimore, C., "SAML 2.0 Bearer Assertion Profiles for OAuth 2.0," draft-ietf-oauth-saml2-bearer-08 (work in progress), August 2011 (TXT).

[I-D.ietf-oauth-v2-bearer] Jones, M., Hardt, D., and D. Recordon, "The OAuth 2.0 Protocol: Bearer Tokens," draft-ietf-oauth-v2-bearer-08 (work in progress), July 2011 (TXT, PDF).

[I-D.ietf-oauth-v2-http-mac] Hammer-Lahav, E., Barth, A., and B. Adida, "HTTP Authentication: MAC Access Authentication," draft-ietf-oauth-v2-http-mac-00 (work in progress), May 2011 (TXT, PDF).

[I-D.ietf-oauth-v2-threatmodel] Lodderstedt, T., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations," draft-ietf-oauth-v2-threatmodel-00 (work in progress), July 2011 (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.

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

Authors' Addresses

TOC

Eran Hammer (editor)
Email: eran@hueniverse.com
URI: http://hueniverse.com

David Recordon
Facebook

Email: dr@fb.com
URI: http://www.davidrecordon.com/

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