Authors: D. Hardt    A. Parecki    T. Lodderstedt
         Hellō       Okta        yes.com
# The OAuth 2.1 Authorization Framework

## Abstract

The OAuth 2.1 authorization framework enables an application to
obtain limited access to a protected resource, either on behalf of a
resource owner by orchestrating an approval interaction between the
resource owner and an authorization service, or by allowing the
application to obtain access on its own behalf. This specification
replaces and obsoletes the OAuth 2.0 Authorization Framework
described in RFC 6749 and the Bearer Token Usage in RFC 6750.

## Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the OAuth Working Group
mailing list (oauth@ietf.org), which is archived at https://
mailarchive.ietf.org/arch/browse/oauth/.

Source for this draft and an issue tracker can be found at https://
github.com/oauth-wg/oauth-v2-1.

## 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 https://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 14 September 2023.

**Table of Contents**

1.  **Introduction**

   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 applications access to restricted
   resources, the resource owner shares their credentials with the
   application. This creates several problems and limitations:

     *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 inherent in passwords.

     *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 often reuse passwords with other unrelated
      services, despite best security practices. This password reuse
      means a vulnerability or exposure in one service may have
      security implications in completely unrelated services.

     *Resource owners cannot revoke access to an individual application
      without revoking access to all third parties, and must do so by
      changing their password.

     *Compromise of any 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. Instead of
   using the resource owner's credentials to access protected
   resources, the client obtains an access token - a credential
   representing a specific set of access attributes such as scope and
   lifetime. Access tokens are issued to 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 their protected photos stored at a photo-
   sharing service (resource server), without sharing their username
   and password with the printing service. Instead, they authenticate
   directly with a server trusted by the photo-sharing service
   (authorization server), which issues the printing service
   delegation- specific credentials (access token).

This separation of concerns also provides the ability to use more advanced user authentication methods such as multi-factor authentication and even passwordless authentication, without any modification to the applications. With all user authentication logic handled by the authorization server, applications don't need to be concerned with the specifics of implementing any particular authentication mechanism. This provides the ability for the authorization server to manage the user authentication policies and even change them in the future without coordinating the changes with applications.

This specification is designed for use with HTTP ([RFC9110]). The use of OAuth over any protocol other than HTTP is out of scope.

Since the publication of the OAuth 2.0 Authorization Framework ([RFC6749]) in October 2012, it has been updated by OAuth 2.0 for Native Apps ([RFC8252]), OAuth Security Best Current Practice ([I-D.ietf-oauth-security-topics]), and OAuth 2.0 for Browser-Based Apps ([I-D.ietf-oauth-browser-based-apps]). The OAuth 2.0 Authorization Framework: Bearer Token Usage ([RFC6750]) has also been updated with ([I-D.ietf-oauth-security-topics]). This Standards Track specification consolidates the information in all of these documents and removes features that have been found to be insecure in [I-D.ietf-oauth-security-topics].

## 1.1.  Roles

OAuth defines four roles:

**"resource owner":**  An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user. This is sometimes abbreviated as "RO".

**"resource server":**  The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens. The resource server is often accessible via an API. This is sometimes abbreviated as "RS".

**"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. This is sometimes abbreviated as "AS".

The interaction between the authorization server and resource server
is beyond the scope of this specification, however several
extensions have been defined to provide an option for
interoperability between resource servers and authorization servers.
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

```
+--------+                               +---------------+
|        |--(1)- Authorization Request ->|   Resource    |
|        |                               |     Owner     |
|        |<-(2)-- Authorization Grant ---|               |
|        |                               +---------------+
|        |
|        |                               +---------------+
|        |--(3)-- Authorization Grant -->| Authorization |
| Client |                               |     Server    |
|        |<-(4)----- Access Token -------|               |
|        |                               +---------------+
|        |
|        |                               +---------------+
|        |--(5)----- Access Token ------>|   Resource    |
|        |                               |     Server    |
|        |<-(6)--- Protected Resource ---|               |
+--------+                               +---------------+
```
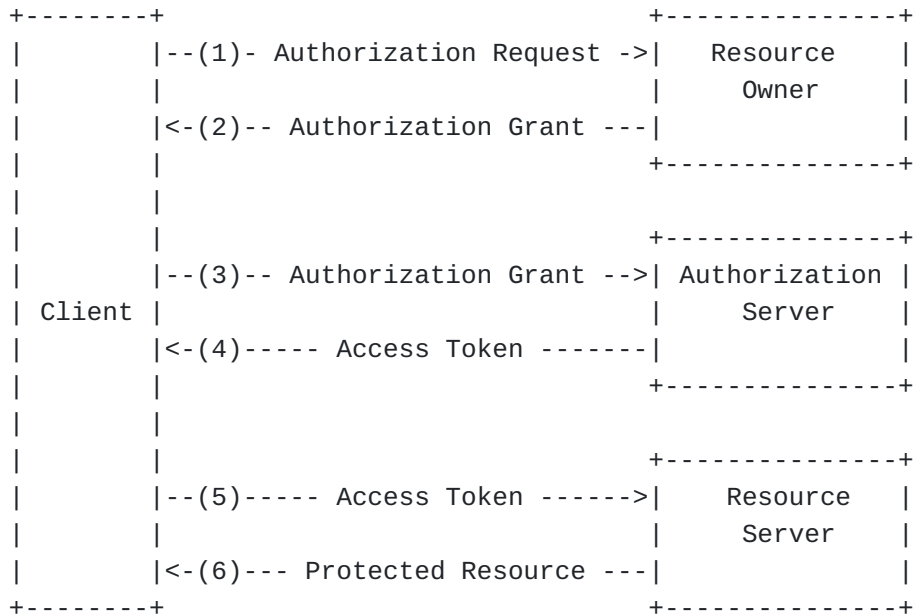
Figure 1: Abstract Protocol Flow

The abstract OAuth 2.1 flow illustrated in Figure 1 describes the
interaction between the four roles and includes the following steps:

1. 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.

2. The client receives an authorization grant, which is a
   credential representing the resource owner's authorization,
   expressed using one of the authorization 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.

3. The client requests an access token by authenticating with the
   authorization server and presenting the authorization grant.

4. The authorization server authenticates the client and validates
      the authorization grant, and if valid, issues an access token.

   5. The client requests the protected resource from the resource
      server and authenticates by presenting the access token.

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

   The preferred method for the client to obtain an authorization grant
   from the resource owner (depicted in steps (1) and (2)) is to use
   the authorization server as an intermediary, which is illustrated in
   [Figure 3](#) in [Section 4.1](#).

## 1.3.  Authorization Grant

   An authorization grant represents the resource owner's authorization
   (to access its protected resources) used by the client to obtain an
   access token. This specification defines three grant types --
   authorization code, refresh token, and client credentials -- as well
   as an extensibility mechanism for defining additional types.

### 1.3.1.  Authorization Code

   An authorization code is a temporary credential used to obtain an
   access token. Instead of the client requesting authorization
   directly from the resource owner, the client directs the resource
   owner to an authorization server (via its user agent, which in turn
   directs the resource owner back to the client with the authorization
   code. The client can then exchange the authorization code for an
   access token.

   Before directing the resource owner back to the client with the
   authorization code, the authorization server authenticates the
   resource owner, and may request the resource owner's consent or
   otherwise inform them of the client's request. Because the resource
   owner only authenticates with the authorization server, the resource
   owner's credentials are never shared with the client, and the client
   does not need to have knowledge of any additional authentication
   steps such as multi-factor authentication or delegated accounts.

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

## 1.3.2.  Refresh Token

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, and may be issued based on properties of the
client, properties of the request, policies within the authorization
server, or any other criteria. If the authorization server issues a
refresh token, it is included when issuing an access token (i.e.,
step (2) in Figure 2).

A refresh token is a string representing the authorization granted
to the client by the resource owner. The string is considered opaque
to the client. The refresh token may be an identifier used to
retrieve the authorization information or may encode this
information into the string itself. Unlike access tokens, refresh
tokens are intended for use only with authorization servers and are
never sent to resource servers.

```
+--------+                                          +---------------+
|        |--(1)------- Authorization Grant --------->|               |
|        |                                          |               |
|        |<-(2)----------- Access Token -------------|               |
|        |                 & Refresh Token          |               |
|        |                                          |               |
|        |                            +----------+  |               |
|        |--(3)---- Access Token ---->|          |  |               |
|        |                            |          |  |               |
|        |<-(4)- Protected Resource --| Resource |  | Authorization |
| Client |                            |  Server  |  |     Server    |
|        |--(5)---- Access Token ---->|          |  |               |
|        |                            |          |  |               |
|        |<-(6)- Invalid Token Error -|          |  |               |
|        |                            +----------+  |               |
|        |                                          |               |
|        |--(7)----------- Refresh Token ----------->|               |
|        |                                          |               |
|        |<-(8)----------- Access Token -------------|               |
+--------+           & Optional Refresh Token       +---------------+
```
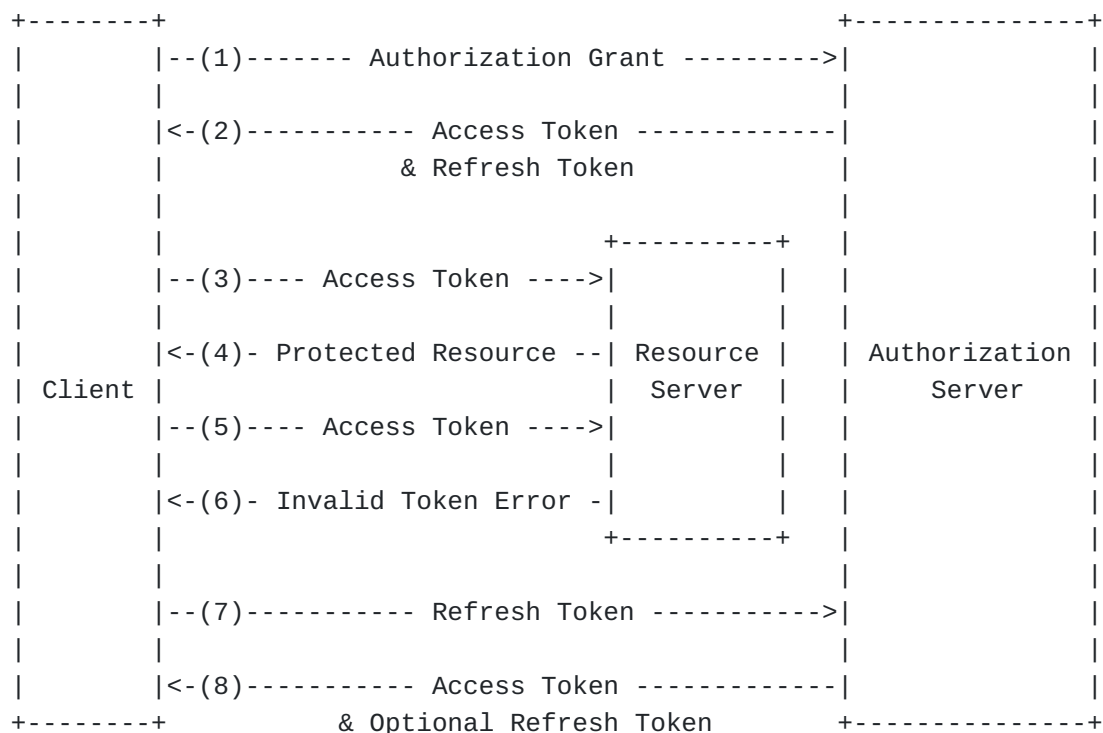
Figure 2: Refreshing an Expired Access Token

The flow illustrated in [Figure 2](#) includes the following steps:

1. The client requests an access token by authenticating with the authorization server and presenting an authorization grant.

2. The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token and optionally a refresh token.

3. The client makes a protected resource request to the resource server by presenting the access token.

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

5. Steps (3) and (4) repeat until the access token expires. If the client knows the access token expired, it skips to step (7); otherwise, it makes another protected resource request.

6. Since the access token is invalid, the resource server returns an invalid token error.

7. The client requests a new access token by presenting the refresh token and providing client authentication if it has been issued credentials. The client authentication requirements are based on the client type and on the authorization server policies.

8. 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).

### 1.3.3.  Client Credentials

The client credentials or other forms of client authentication (e.g. a private key used to sign a JWT, as described in [[RFC7523](#)]) 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 when the client is requesting access to protected resources based on an authorization previously arranged with the authorization server.

### 1.4.  Access Token

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 considered opaque to the client, even if it has a structure. Depending on the authorization server, the access

token string may be parseable by the resource server, such as when using the JSON Web Token Profile for Access Tokens ([RFC9068]).

Access 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 be used by the RS to retrieve the authorization information, or the token may self-contain the authorization information in a verifiable manner (i.e., a token string consisting of a signed data payload). One example of a token retrieval mechanism is Token Introspection [RFC7662], in which the RS calls an endpoint on the AS to validate the token presented by the client. One example of a structured token format is [RFC9068], a method of encoding access token data as a JSON Web Token [RFC7519].

Additional authentication credentials, which are beyond the scope of this specification, may be required in order for the client to use an access token. This is typically referred to as a sender-constrained access token, such as DPoP [I-D.ietf-oauth-dpop] and Mutual TLS Access Tokens [RFC8705].

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 may be extended beyond what is described in this specification.

Access tokens (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.

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

## 1.5.  Communication security

Implementations MUST use a mechanism to provide communication authentication, integrity and confidentiality such as Transport-Layer Security [RFC8446], to protect the exchange of clear-text

credentials and tokens either in the content or in header fields
from eavesdropping, tampering, and message forgery (eg. see
Section 2.4.1, Section 7.6, Section 3.2, and Section 5.2).

OAuth URLs MUST use the https scheme except for loopback interface
redirect URIs, which MAY use the http scheme. When using https, TLS
certificates MUST be checked according to [RFC9110]. At the time of
this writing, TLS version 1.3 [RFC8446] is the most recent version.

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

The identification of the TLS versions and algorithms is outside the
scope of this specification. Refer to [BCP195] for up to date
recommendations on transport layer security, and to the relevant
specifications for certificate validation and other security
considerations.

## 1.6.  HTTP Redirections

This specification makes extensive use of HTTP redirections, in
which the client or the authorization server directs the resource
owner's user agent to another destination. While the examples in
this specification show the use of the HTTP 302 status code, any
other method available via the user agent to accomplish this
redirection, with the exception of HTTP 307, is allowed and is
considered to be an implementation detail. See Section 7.5.2 for
details.

## 1.7.  Interoperability

OAuth 2.1 provides a rich authorization framework with well-defined
security properties.

This specification leaves a few required components partially or
fully undefined (e.g., client registration, authorization server
capabilities, endpoint discovery). Some of these behaviors are
defined in optional extensions which implementations can choose to
use, such as:

  *[RFC8414]: Authorization Server Metadata, defining an endpoint
   clients can use to look up the information needed to interact
   with a particular OAuth server

  *[RFC7591]: Dynamic Client Registration, providing a mechanism for
   programmatically registering clients with an authorization server

  *[RFC7592]: Dynamic Client Management, providing a mechanism for
   updating dynamically registered client information

*[RFC7662]: Token Introspection, defining a mechanism for resource
    servers to obtain information about access tokens

   Please refer to Appendix C for a list of current known extensions at
   the time of this publication.

## 1.8.  Compatibility with OAuth 2.0

   OAuth 2.1 is compatible with OAuth 2.0 with the extensions and
   restrictions from known best current practices applied.
   Specifically, features not specified in OAuth 2.0 core, such as
   PKCE, are required in OAuth 2.1. Additionally, some features
   available in OAuth 2.0, such as the Implicit or Resource Owner
   Credentials grant types, are not specified in OAuth 2.1.
   Furthermore, some behaviors allowed in OAuth 2.0 are restricted in
   OAuth 2.1, such as the strict string matching of redirect URIs
   required by OAuth 2.1.

   See Section 10 for more details on the differences from OAuth 2.0.

## 1.9.  Notational Conventions

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and
   "OPTIONAL" in this document are to be interpreted as described in
   BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all
   capitals, as shown here.

   This specification uses the Augmented Backus-Naur Form (ABNF)
   notation of [RFC5234]. Additionally, the rule URI-reference is
   included from "Uniform Resource Identifier (URI): Generic Syntax"
   [RFC3986].

   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".

   The term "content" is to be interpreted as described in Section 6.4
   of [RFC9110].

   The term "user agent" is to be interpreted as described in Section
   3.5 of [RFC9110].

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

## 2.  Client Registration

Before initiating the protocol, the client must establish its
registration 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 the client
developer manually registering the client at the authorization
server's website after creating an account and agreeing to the
service's Terms of Service, or by using Dynamic Client Registration
([RFC7591]).

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., redirect 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 SHALL:

  *specify the client type as described in Section 2.1,

  *provide client details needed by the grant type in use, such as
   redirect URIs as described in Section 2.3, and

  *include any other information required by the authorization
   server (e.g., application name, website, description, logo image,
   the acceptance of legal terms).

Dynamic Client Registration ([RFC7591]) defines a common general
data model for clients that may be used even with manual client
registration.

## 2.1.  Client Types

OAuth 2.1 defines two client types based on their ability to
authenticate securely with the authorization server.

**"confidential":**  Clients that have credentials with the AS are
   designated as "confidential clients"

**"public":**  Clients without credentials are called "public clients"

Any clients with credentials MUST take precautions to prevent
leakage and abuse of their credentials.

Client authentication allows an Authorization Server to ensure it is
interacting with a certain client (identified by its client_id) in

an OAuth flow. The Authorization Server might make policy decisions about things such as whether to prompt the user for consent on every authorization or only the first based on the confidence that the Authorization Server is actually communicating with the legitimate client.

Whether and how an Authorization Server validates the identity of a client or the party providing/operating this client is out of scope of this specification. Authorization servers SHOULD consider the level of confidence in a client's identity when deciding whether they allow a client access to more sensitive resources and operations such as the Client Credentials grant type and how often to prompt the user for consent.

A single client_id SHOULD NOT be treated as more than one type of client.

This specification has been designed around the following client profiles:

**"web application":**  A web application is a 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 tokens issued to the client are stored on the web server and are not exposed to or accessible by the resource owner.

**"browser-based application":**  A browser-based application is a 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. If such applications wish to use client credentials, it is recommended to utilize the backend for frontend pattern. 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 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. Dynamically issued access tokens and refresh tokens can receive an acceptable level of protection. On some platforms, these credentials are protected from other applications residing on the same device. If such applications wish to use client credentials, it is recommended to utilize the backend for frontend pattern, or issue the

credentials at runtime using Dynamic Client Registration
([RFC7591]).

## 2.2.  Client Identifier

Every client is identified in the context of an authorization server
by a client identifier -- a unique string representing the
registration information provided by the client. While the
Authorization Server typically issues the client identifier itself,
it may also serve clients whose client identifier was created by a
party other than the Authorization Server. The client identifier is
not a secret; it is exposed to the resource owner and MUST NOT be
used alone for client authentication. The client identifier is
unique in the context of an authorization server.

The client identifier string size is left undefined by this
specification. The client should avoid making assumptions about the
identifier size. The authorization server SHOULD document the size
of any identifier it issues.

If the authorization server supports clients with client identifiers
issued by parties other than the authorization server, the
authorization server SHOULD take precautions to avoid clients
impersonating resource owners as described in Section 7.4.

## 2.3.  Client Redirection Endpoint

The client redirection endpoint (also referred to as "redirect
endpoint") is the URI of the client that the authorization server
redirects the user agent back to after completing its interaction
with the resource owner.

The authorization server redirects the user agent to one of the
client's redirection endpoints previously established with the
authorization server during the client registration process.

The redirect 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 query component ([WHATWG.URL]), which
MUST be retained when adding additional query parameters. The
endpoint URI MUST NOT include a fragment component.

### 2.3.1.  Registration Requirements

Authorization servers MUST require clients to register their
complete redirect URI (including the path component). Authorization
servers MUST reject authorization requests that specify a redirect
URI that doesn't exactly match one that was registered, with an
exception for loopback redirects, where an exact match is required
except for the port URI component.

The authorization server MAY allow the client to register multiple redirect URIs.

Registration may happen out of band, such as a manual step of configuring the client information at the authorization server, or may happen at runtime, such as in the initial POST in Pushed Authorization Requests [RFC9126].

For private-use URI scheme-based redirect URIs, authorization servers SHOULD enforce the requirement in Section 8.4.3 that clients use schemes that are reverse domain name based. At a minimum, any private-use URI scheme that doesn't contain a period character (.) SHOULD be rejected.

In addition to the collision-resistant properties, this can help to prove ownership in the event of a dispute where two apps claim the same private-use URI scheme (where one app is acting maliciously). For example, if two apps claimed com.example.app, the owner of example.com could petition the app store operator to remove the counterfeit app. Such a petition is harder to prove if a generic URI scheme was used.

Clients MUST NOT expose URLs that forward the user's browser to arbitrary URIs obtained from a query parameter ("open redirector"), as described in Section 7.13. Open redirectors can enable exfiltration of authorization codes and access tokens.

The client MAY use the state request parameter to achieve per-request customization if needed rather than varying the redirect URI per request.

Without requiring registration of redirect URIs, attackers can use the authorization endpoint as an open redirector as described in Section 7.13.

### 2.3.2. Multiple Redirect URIs

If multiple redirect URIs have been registered, the client MUST include a redirect URI with the authorization request using the redirect_uri request parameter.

### 2.3.3. Preventing CSRF Attacks

Clients MUST prevent Cross-Site Request Forgery (CSRF) attacks. In this context, CSRF refers to requests to the redirection endpoint that do not originate at the authorization server, but a malicious third party (see Section 4.4.1.8. of [RFC6819] for details). Clients that have ensured that the authorization server supports the code_challenge parameter MAY rely on the CSRF protection provided by that mechanism. In OpenID Connect flows, validating the nonce

parameter provides CSRF protection. Otherwise, one-time use CSRF
tokens carried in the state parameter that are securely bound to the
user agent MUST be used for CSRF protection (see Section 7.10).

### 2.3.4.  Preventing Mix-Up Attacks

In order to prevent mix-up attacks, clients MUST only process
redirect responses of the authorization server they sent the
respective request to and from the same user agent this
authorization request was initiated with. Clients MUST store the
authorization server they sent an authorization request to and bind
this information to the user agent and check that the authorization
response was received from the correct authorization server. Clients
MUST ensure that the subsequent access token request, if applicable,
is sent to the same authorization server. Clients SHOULD use
distinct redirect URIs for each authorization server as a means to
identify the authorization server a particular response came from.

### 2.3.5.  Invalid Endpoint

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

### 2.3.6.  Endpoint Content

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 redirect URI and the credentials (e.g.
authorization code) it contains. Additionally, the request URL
containing the authorization code may be sent in the HTTP Referer
header to any embedded images, stylesheets and other elements loaded
in the page.

The client SHOULD NOT include any third-party scripts (e.g., third-
party analytics, social plug-ins, ad networks) in the redirection
endpoint response. Instead, it SHOULD extract the credentials from
the URI and redirect the user agent again to another endpoint
without exposing the credentials (in the URI or elsewhere). If
third-party scripts are included, the client MUST ensure that its
own scripts (used to extract and remove the credentials from the
URI) will execute first.

## 2.4. Client Authentication

The authorization server MUST only rely on client authentication if
the process of issuance/registration and distribution of the
underlying credentials ensures their confidentiality.

If the client is confidential, the authorization server MAY accept
any form of client authentication meeting its security requirements
(e.g., password, public/private key pair).

It is RECOMMENDED to use asymmetric (public-key based) methods for
client authentication such as mTLS [RFC8705] or a JWT [RFC7523].
When asymmetric methods for client authentication are used,
authorization servers do not need to store sensitive symmetric keys,
making these methods more robust against a number of attacks.

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 redirect URI or
enlisting the resource owner to confirm identity. A valid redirect
URI is not sufficient to verify the client's identity when asking
for resource owner authorization but can be used to prevent
delivering credentials to a counterfeit client after obtaining
resource owner authorization.

The client MUST NOT use more than one authentication method in each
request to prevent a conflict of which authentication mechanism is
authoritative for the request.

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

The privileges an authorization server associates with a certain
client identity MUST depend on the assessment of the overall process
for client identification and client credential lifecycle
management. See Section 7.2 for additional details.

### 2.4.1. Client Secret

Clients in possession of a client secret, sometimes known as a
client password, MAY use the HTTP Basic authentication scheme as
defined in Section 11 of [RFC9110] to authenticate with the
authorization server. The client identifier is encoded using the
application/x-www-form-urlencoded encoding algorithm per Appendix B,
and the encoded value is used as the username; the client secret is
encoded using the same algorithm and used as the password. The
authorization server MUST support the HTTP Basic authentication
scheme for authenticating clients that were issued a client secret.

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

Authorization: Basic czZCaGRSa3F0Mzo3RmpmcDBaQnIxS3REUmJuZlZkbUl3

In addition to that, the authorization server MAY support including the client credentials in the request content 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.

Including the client credentials in the request content 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 content and MUST NOT be included in the request URI.

For example, a request to refresh an access token (Section 4.3) using the content parameters (with extra line breaks for display purposes only):

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

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

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

### 2.4.2.  Other Authentication Methods

The authorization server MAY support any suitable 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.

Some additional authentication methods such as mTLS [RFC8705] and Private Key JWT [RFC7523] are defined in the "OAuth Token Endpoint Authentication Methods" registry, and may be useful as generic client authentication methods beyond the specific use of protecting the token endpoint.

## 2.5.  Unregistered Clients

This specification does not require that clients be registered with
the authorization server. However, the use of unregistered clients
is beyond the scope of this specification and requires additional
security analysis and review of its interoperability impact.

## 3.  Protocol Endpoints

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

  *Authorization endpoint - used by the client to obtain
   authorization from the resource owner via user agent redirection.

  *Token endpoint - used by the client to exchange an authorization
   grant for an access token, typically with client authentication.

As well as one client endpoint:

  *Redirection endpoint - used by the authorization server to return
   responses containing authorization credentials 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

The authorization endpoint is used to interact with the resource
owner and obtain an authorization grant. The authorization server
MUST first authenticate the resource owner. The way in which the
authorization server authenticates the resource owner (e.g.,
username and password login, passkey, federated login, or by using
an established session) 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,
or in the authorization server's metadata document ([RFC8414]).

The endpoint URI MAY include an "application/x-www-form-urlencoded"
formatted query component [WHATWG.URL], which MUST be retained when
adding additional query parameters. The endpoint URI MUST NOT
include a fragment component.

The authorization server MUST support the use of the HTTP GET method
Section 9.3.1 of [RFC9110] for the authorization endpoint and MAY
support the POST method (Section 9.3.3 of RFC9110) as well.

The authorization server MUST ignore unrecognized request
parameters.

Request and response parameters defined by this specification MUST
NOT be included more than once. Parameters sent without a value MUST
be treated as if they were omitted from the request.

An authorization server that redirects a request potentially
containing user credentials MUST avoid forwarding these user
credentials accidentally (see Section 7.5.2 for details).

## 3.2. Token Endpoint

The token endpoint is used by the client to obtain an access token
using a grant such as those described in Section 4 and Section 4.3.

The means through which the client obtains the location of the token
endpoint are beyond the scope of this specification, but the
location is typically provided in the service documentation and
configured during development of the client, or provided in the
authorization server's metadata document ([RFC8414]) and fetched
programmatically at runtime.

The endpoint URI MAY include an application/x-www-form-urlencoded
formatted query component ([WHATWG.URL]) and MUST NOT include a
fragment component.

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

The authorization server MUST ignore unrecognized request
parameters.

Parameters sent without a value MUST be treated as if they were
omitted from the request. Request and response parameters defined by
this specification MUST NOT be included more than once.

Authorization servers that wish to support browser-based
applications (applications running exclusively in client-side
JavaScript without access to a supporting backend server) will need
to ensure the token endpoint supports the necessary Cross-Origin
Resource Sharing (CORS) headers. If the authorization server
provides additional endpoints to the application, such as metadata
URLs, dynamic client registration, revocation, introspection,
discovery or user info endpoints, these endpoints may also be
accessed by the browser-based application, and will also need to
have the CORS headers defined to allow access. See
[I-D.ietf-oauth-browser-based-apps] for further details.

### 3.2.1.  Client Authentication

Confidential clients MUST authenticate with the authorization server
as described in Section 2.4 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 adds an
   additional layer of security when an authorization code is
   transmitted to the redirection endpoint over an insecure channel.

  *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.

### 3.2.2.  Token Request

The client makes a request to the token endpoint by sending the
following parameters using the application/x-www-form-urlencoded
format per Appendix B with a character encoding of UTF-8 in the HTTP
request content:

**"client_id":**  REQUIRED, if the client is not authenticating with the
    authorization server as described in Section 3.2.1.

**"grant_type":**  REQUIRED. Identifier of the grant type the client
    uses with the particular token request. This specification
    defines the values authorization_code, refresh_token, and
    client_credentials. The grant type determines the further
    parameters required or supported by the token request. The
    details of those grant types are defined below.

Confidential clients MUST authenticate with the authorization server
as described in Section 3.2.1.

For example, the client makes the following HTTP request (with extra
line breaks for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&code_verifier=3641a2d12d66101249cdf7a79c000c1f8c05d2aafcf14bf146497bed
```

The authorization server MUST:

  *require client authentication for confidential clients (or
   clients with other authentication requirements),

  *authenticate the client if client authentication is included

Further grant type specific processing rules apply and are specified
with the respective grant type.

## 3.2.2.1.  Access Token Scope

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 in the token
response ([Section 3.2.3](#)) 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).

### 3.2.3. Token Response

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token.

If the request client authentication failed or is invalid, the authorization server returns an error response as described in Section 3.2.3.1.

The authorization server issues an access token and optional refresh token by creating an HTTP response content using the application/json media type as defined by [RFC8259] with the following parameters and an HTTP 200 (OK) status code:

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

**"token_type":**  REQUIRED. The type of the access token issued as described in Section 5.1. 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":**  RECOMMENDED, if identical to the scope requested by the client; otherwise, REQUIRED. The scope of the access token as described by Section 3.2.2.1.

**"refresh_token":**  OPTIONAL. The refresh token, which can be used to obtain new access tokens based on the grant passed in the corresponding token request.

Authorization servers SHOULD determine, based on a risk assessment and their own policies, whether to issue refresh tokens to a certain client. If the authorization server decides not to issue refresh tokens, the client MAY obtain new access tokens by starting the OAuth flow over, for example initiating a new authorization code request. In such a case, the authorization server may utilize cookies and persistent grants to optimize the user experience.

If refresh tokens are issued, those refresh tokens MUST be bound to the scope and resource servers as consented by the resource owner. This is to prevent privilege escalation by the legitimate client and reduce the impact of refresh token leakage.

The parameters are serialized into a JavaScript Object Notation (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 (see Section 5.2 of [RFC9111]) with a value
of no-store in any response containing tokens, credentials, or other
sensitive information.

For example:

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

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"Bearer",
  "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.

### 3.2.3.1.  Error Response

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

**"error":**  REQUIRED. A single ASCII [USASCII] error code from the
   following:

   **"invalid_request":**  The request is missing a required parameter,
      includes an unsupported parameter value (other than grant
      type), repeats a parameter, includes multiple credentials,
      utilizes more than one mechanism for authenticating the
      client, contains a code_verifier although no code_challenge
      was sent in the authorization request, 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) or refresh
token is invalid, expired, revoked, does not match the
redirect 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.

Values for the error parameter MUST NOT include characters
outside the set %x20-21 / %x23-5B / %x5D-7E.

**"error_description":**  OPTIONAL. Human-readable ASCII [USASCII] text
providing additional information, used to assist the client
developer in understanding the error that occurred. Values for
the error_description parameter MUST NOT include characters
outside the set %x20-21 / %x23-5B / %x5D-7E.

**"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. Values for
the error_uri parameter MUST conform to the URI-reference syntax
and thus MUST NOT include characters outside the set %x21 /
%x23-5B / %x5D-7E.

The parameters are included in the content of the HTTP response
using the application/json media type as defined by [RFC7159]. 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
Cache-Control: no-store

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

## 4.  Grant Types

To request an access token, the client obtains authorization from
the resource owner. This specification defines the following
authorization grant types:

   *authorization code

   *client credentials, and

   *refresh token

It also provides an extension mechanism for defining additional
grant types.

## 4.1.  Authorization Code Grant

The authorization code grant type is used to obtain both access
tokens and refresh tokens.

The grant type uses the additional authorization endpoint to let the
authorization server interact with the resource owner in order to
get consent for resource access.

Since this is a redirect-based flow, the client must be capable of
initiating the flow with the resource owner's user agent (typically
a web browser) and capable of being redirected back to from the
authorization server.

```
+----------+
| Resource |
|   Owner  |
+----------+
     ^
     |
     |
+-----|----+          Client Identifier      +---------------+
| .---+---------(1)-- & Redirection URI ---->|               |
| |   |    |    |                            |               |
| |   '---------(2)-- User authenticates --->|               |
| | User-   |                                | Authorization |
| | Agent   |                                |     Server    |
| |   |     |                                |               |
| |   .---------(3)-- Authorization Code ---<|               |
+-|----|---+                                 +---------------+
  |    |                                           ^      v
  |    |                                           |      |
  ^    v                                           |      |
+---------+                                        |      |
|         |>---(4)-- Authorization Code ---------' |
|  Client |          & Redirection URI             |
|         |                                        |
|         |<---(5)----- Access Token -------------------'
+---------+         (w/ Optional Refresh Token)
```
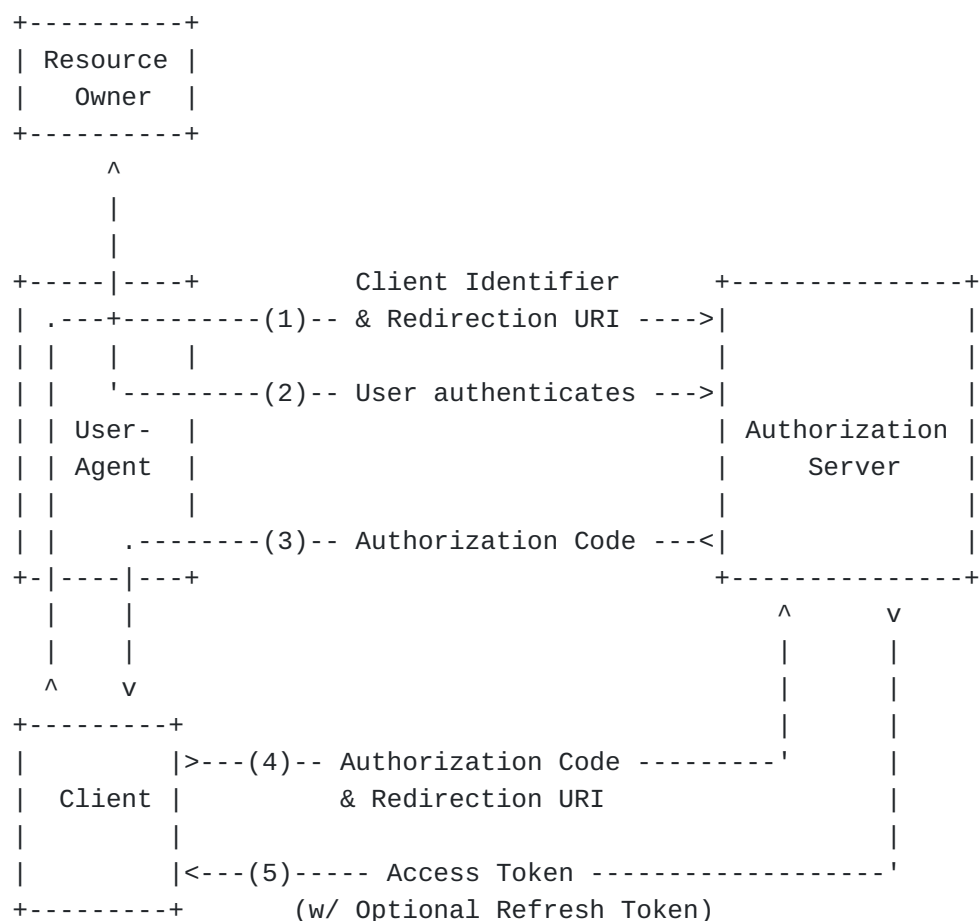
                    Figure 3: Authorization Code Flow

   The flow illustrated in Figure 3 includes the following steps:

   (1) The client initiates the flow by directing the resource owner's
   user agent to the authorization endpoint. The client includes its
   client identifier, code challenge (derived from a generated code
   verifier), optional requested scope, optional local state, and a
   redirect URI to which the authorization server will send the user
   agent back once access is granted (or denied).

   (2) 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.

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

   (4) The client requests an access token from the authorization
   server's token endpoint by including the authorization code received

in the previous step, and including its code verifier. When making
the request, the client authenticates with the authorization server
if it can. The client includes the redirect URI used to obtain the
authorization code for verification.

(5) The authorization server authenticates the client when possible,
validates the authorization code, validates the code verifier, and
ensures that the redirect URI received matches the URI used to
redirect the client in step (3). If valid, the authorization server
responds back with an access token and, optionally, a refresh token.

## 4.1.1.  Authorization Request

To begin the authorization request, the client builds the
authorization request URI by adding parameters to the authorization
server's authorization endpoint URI. The client will eventually
redirect the user agent to this URI to initiate the request.

Clients use a unique secret per authorization request to protect
against authorization code injection and CSRF attacks. The client
first generates this secret, which it can use at the time of
redeeming the authorization code to prove that the client using the
authorization code is the same client that requested it.

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, per Appendix B:

**"response_type":**  REQUIRED. The authorization endpoint supports
   different sets of request and response pameters. The client
   determines the type of flow by using a certain response_type
   value. This specification defines the value code, which must be
   used to signal that the client wants to use the authorization
   code flow.

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.

Some extension response types are defined by ([OpenID]).

If an authorization request is missing the response_type parameter,
or if the response type is not understood, the authorization server
MUST return an error response as described in Section 4.1.2.1.

**"client_id":**
    REQUIRED. The client identifier as described in
    [Section 2.2](#).

**"code_challenge":**  REQUIRED or RECOMMENDED (see [Section 7.6](#)). Code
    challenge.

**"code_challenge_method":**  OPTIONAL, defaults to plain if not present
    in the request. Code verifier transformation method is S256 or
    plain.

**"redirect_uri":**  OPTIONAL. As described in [Section 2.3](#).

**"scope":**  OPTIONAL. The scope of the access request as described by
    [Section 3.2.2.1](#).

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

The code_verifier is a unique high-entropy cryptographically random
string generated for each authorization request, using the
unreserved characters [A-Z] / [a-z] / [0-9] / "-" / "." / "_" / "~",
with a minimum length of 43 characters and a maximum length of 128
characters.

The client stores the code_verifier temporarily, and calculates the
code_challenge which it uses in the authorization request.

ABNF for code_verifier is as follows.

```
code-verifier = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

Clients SHOULD use code challenge methods that do not expose the
code_verifier in the authorization request. Otherwise, attackers
that can read the authorization request (cf. Attacker A4 in
[[I-D.ietf-oauth-security-topics](#)]) can break the security provided by
this mechanism. Currently, S256 is the only such method.

NOTE: The code verifier SHOULD have enough entropy to make it
impractical to guess the value. It is RECOMMENDED that the output of
a suitable random number generator be used to create a 32-octet
sequence. The octet sequence is then base64url-encoded to produce a
43-octet URL-safe string to use as the code verifier.

The client then creates a code_challenge derived from the code
verifier by using one of the following transformations on the code
verifier:

S256

  code_challenge = BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))

plain

  code_challenge = code_verifier

If the client is capable of using S256, it MUST use S256, as S256 is
Mandatory To Implement (MTI) on the server. Clients are permitted to
use plain only if they cannot support S256 for some technical
reason, for example constrained environments that do not have a
hashing function available, and know via out-of-band configuration
or via Authorization Server Metadata ([RFC8414]) that the server
supports plain.

ABNF for code_challenge is as follows.

```
code-challenge = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

The properties code_challenge and code_verifier are adopted from the
OAuth 2.0 extension known as "Proof-Key for Code Exchange", or PKCE
([RFC7636]) where this technique was originally developed.

Authorization servers MUST support the code_challenge and
code_verifier parameters.

Clients MUST use code_challenge and code_verifier and authorization
servers MUST enforce their use except under the conditions described
in Section 7.6. In this case, using and enforcing code_challenge and
code_verifier as described in the following is still RECOMMENDED.

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.

The client directs the resource owner to the constructed URI using
an HTTP redirection, 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 (with extra line breaks for display purposes only):

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz
    &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
    &code_challenge=6fdkQaPm51l13DSukcAH3Mdx7_ntecHYd1vi3n0hMZY
    &code_challenge_method=S256 HTTP/1.1
Host: server.example.com
```

   The authorization server validates the request to ensure that all
   required parameters are present and valid.

   In particular, the authorization server MUST validate the
   redirect_uri in the request if present, ensuring that it matches one
   of the registered redirect URIs previously established during client
   registration (Section 2). When comparing the two URIs the
   authorization server MUST using simple character-by-character string
   comparison as defined in [RFC3986], Section 6.2.1.

   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 redirect URI using an HTTP
   redirection response, or by other means available to it via the user
   agent.

## 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
   redirect URI using the application/x-www-form-urlencoded format, per
   Appendix B:

   **"code"**:  REQUIRED. The authorization code is generated by the
      authorization server and opaque to the client. 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 access tokens and refresh tokens previously
      issued based on that authorization code. The authorization code
      is bound to the client identifier, code challenge and redirect
      URI.

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

   **"iss"**:  OPTIONAL. The identifier of the authorization server which
      the client can use to prevent mixup attacks, if the client

interacts with more than one authorization server. See [RFC9207]
for additional details on when this parameter is necessary, and
how the client can use it to prevent mixup attacks.

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&iss=https%3A%2F%2Fauthorization-server.example.com
```

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.

The authorization server MUST associate the code_challenge and
code_challenge_method values with the issued authorization code so
the code challenge can be verified later.

The exact method that the server uses to associate the
code_challenge with the issued code is out of scope for this
specification. The code challenge could be stored on the server and
associated with the code there. The code_challenge and
code_challenge_method values may be stored in encrypted form in the
code itself, but the server MUST NOT include the code_challenge
value in a response parameter in a form that entities other than the
AS can extract.

Clients MUST prevent injection (replay) of authorization codes into
the authorization response by attackers. Using code_challenge and
code_verifier prevents injection of authorization codes since the
authorization server will reject a token request with a mismatched
code_verifier. See Section 7.6 for more details.

### 4.1.2.1.  Error Response

If the request fails due to a missing, invalid, or mismatching
redirect 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
redirect URI.

An AS MUST reject requests without a code_challenge from public
clients, and MUST reject such requests from other clients unless
there is reasonable assurance that the client mitigates
authorization code injection in other ways. See Section 7.6 for
details.

If the server does not support the requested code_challenge_method transformation, the authorization endpoint MUST return the authorization error response with error value set to invalid_request. The error_description or the response of error_uri SHOULD explain the nature of error, e.g., transform algorithm not supported.

If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirect URI, the authorization server informs the client by adding the following parameters to the query component of the redirect URI using the application/x-www-form-urlencoded format, per [Appendix B](#):

**"error"**:  REQUIRED. A single ASCII [[USASCII](#)] error code from the following:

> **"invalid_request"**:  The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, 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 that prevented it from fulfilling the request. (This error code is needed because a 500 Internal Server Error HTTP status code cannot be returned to the client via an HTTP redirect.)

> **"temporarily_unavailable"**:  The authorization server is currently unable to handle the request due to a temporary overloading or maintenance of the server. (This error code is needed because a 503 Service Unavailable HTTP status code cannot be returned to the client via an HTTP redirect.)

> Values for the error parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

**"error_description"**:  OPTIONAL. Human-readable ASCII [[USASCII](#)] text providing additional information, used to assist the client developer in understanding the error that occurred. Values for

the error_description parameter MUST NOT include characters
outside the set %x20-21 / %x23-5B / %x5D-7E.

"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. Values for
the error_uri parameter MUST conform to the URI-reference syntax
and thus MUST NOT include characters outside the set %x21 /
%x23-5B / %x5D-7E.

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

"iss":  OPTIONAL. The identifier of the authorization server. See
Section 4.1.2 above for details.

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&iss=https%3A%2F%2Fauthorization-server.example.com
```

### 4.1.3.  Token Endpoint Extension

The authorization grant type is identified at the token endpoint
with the grant_type value of authorization_code.

If this value is set, the following additional token request
parameters beyond Section 3.2.2 are required:

"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, in which case their values MUST be identical. If
no redirect_uri was included in the authorization request, this
parameter is OPTIONAL.

"code_verifier":  REQUIRED, if the code_challenge parameter was
included in the authorization request. MUST NOT be used
otherwise. The original code verifier string.

For example, the client makes the following HTTP request (with extra
line breaks for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&code_verifier=3641a2d12d66101249cdf7a79c000c1f8c05d2aafcf14bf146497bed
```

In addition to the processing rules in Section 3.2.2, the
authorization server MUST:

  *ensure that the authorization code was issued to the
   authenticated confidential client, or if the client is public,
   ensure that the code was issued to client_id in the request,

  *verify that the authorization code is valid,

  *verify that the code_verifier parameter is present if and only if
   a code_challenge parameter was present in the authorization
   request,

  *if a code_verifier is present, verify the code_verifier by
   calculating the code challenge from the received code_verifier
   and comparing it with the previously associated code_challenge,
   after first transforming it according to the
   code_challenge_method method specified by the client, 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
   that their values are identical.

  *If there was no code_challenge in the authorization request
   associated with the authorization code in the token request, the
   authorization server MUST reject the token request.

## 4.2.  Client Credentials Grant

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 that have 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
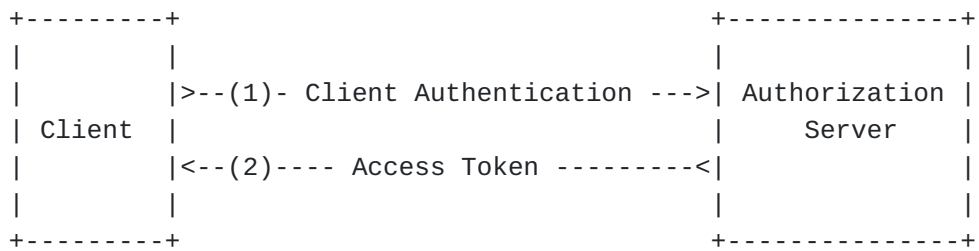clients.

```
+---------+                              +---------------+
|         |                              |               |
|         |>--(1)- Client Authentication --->| Authorization |
| Client  |                              |     Server    |
|         |<--(2)---- Access Token --------<|               |
|         |                              |               |
+---------+                              +---------------+
```

                  Figure 4: Client Credentials Grant

   The use of the client credentials grant illustrated in Figure 4
   includes the following steps:

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

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

### 4.2.1.  Token Endpoint Extension

   The authorization grant type is identified at the token endpoint
   with the grant_type value of client_credentials.

   If this value is set, the following additional token request
   parameters beyond Section 3.2.2 are supported:

   **"scope":**  OPTIONAL. The scope of the access request as described by
        Section 3.2.2.1.

   For example, the client makes the following HTTP request using
   transport-layer security (with extra line breaks for display
   purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
```

   The authorization server MUST authenticate the client.

### 4.3.  Refresh Token Grant

   The refresh token is a credential issued by the authorization server
   to a client, which can be used to obtain new (fresh) access tokens
   based on an existing grant. The client uses this option either
   because the previous access token has expired or the client
   previously obtained an access token with a scope more narrow than

approved by the respective grant and later requires an access token with a different scope under the same grant.

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.

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 issue sender-constrained refresh tokens or use refresh token rotation as described in (#refresh-token-endpoint-extension).

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

### 4.3.1. Token Endpoint Extension

The authorization grant type is identified at the token endpoint with the grant_type value of refresh_token.

If this value is set, the following additional parameters beyond Section 3.2.2 are required/supported:

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

**"scope":**  OPTIONAL. The scope of the access request as described by Section 3.2.2.1. 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 to which it was issued. Confidential clients 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 (with extra line breaks for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
```

In addition to the processing rules in Section 3.2.2, the
authorization server MUST:

  *validate the refresh token.

Authorization servers SHOULD utilize one of these methods to detect
refresh token replay by malicious actors for public clients:

  *Sender-constrained refresh tokens: the authorization server
   cryptographically binds the refresh token to a certain client
   instance by utilizing [I-D.ietf-oauth-token-binding], [RFC8705],
   [I-D.ietf-oauth-dpop], or another suitable method.

  *Refresh token rotation: the authorization server issues a new
   refresh token with every access token refresh response. The
   previous refresh token is invalidated but information about the
   relationship is 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 cannot determine
   which party submitted the invalid refresh token, but it will
   revoke the active refresh token as well as the access
   authorization grant associated with it. This stops the attack at
   the cost of forcing the legitimate client to obtain a fresh
   authorization grant.

Implementation note: the grant to which a refresh token belongs may
be encoded into the refresh token itself. This can enable an
authorization server to efficiently determine the grant to which a
refresh token belongs, and by extension, all refresh tokens that
need to be revoked. Authorization servers MUST ensure the integrity
of the refresh token value in this case, for example, using
signatures.

### 4.3.2.  Refresh Token Response

If valid and authorized, the authorization server issues an access
token as described in Section 3.2.3.

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.

Authorization servers MAY revoke refresh tokens automatically in case of a security event, such as:

  *password change

  *logout at the authorization server

Refresh tokens SHOULD expire if the client has been inactive for some time, i.e., the refresh token has not been used to obtain new access tokens for some time. The expiration time is at the discretion of the authorization server. It might be a global value or determined based on the client policy or the grant associated with the refresh token (and its sensitivity).

## 4.4.  Extension Grants

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 the Device Authorization Grant as defined by [RFC8628] after the user has authorized the client on a separate device, the client makes the following HTTP request (with extra line breaks for display purposes only):

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Adevice_code
&device_code=GmRhmhcxhwEzkoEqiMEg_DnyEysNkuNhszIySk9eS
&client_id=C409020731
```

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in Section 3.2.3. If the request failed client authentication or is invalid, the authorization server returns an error response as described in Section 3.2.3.1.

## 5.  Accessing Protected Resources

The client accesses protected resources by presenting the access token to the resource server. The resource server MUST validate the

access token and ensure that 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. For example, when the resource server and authorization server are colocated or are part of the same system, they may share a database or other storage; when the two components are operated independently, they may use Token Introspection [RFC7662] or a structured access token format such as a JWT [RFC9068].

The method in which the client utilizes the access token to access protected resources at 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 [RFC7235] with an authentication scheme defined by the specification of the access token type used, such as Bearer, defined below.

## 5.1. Access Token Types

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 this specification is utilized by simply including the access token string in the request:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: Bearer mF_9.B5f-4.1JqM
```

The above example is provided for illustration purposes only.

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.

## 5.2. Bearer Tokens

A Bearer Token is a security token with the property that any party in possession of the token (a "bearer") can use the token in any way that any other party in possession of it can. Using a Bearer Token does not require a bearer to prove possession of cryptographic key material (proof-of-possession).

Bearer Tokens may be enhanced with proof-of-possession specifications such as mTLS [RFC8705] to provide proof-of-possession characteristics.

To protect against access token disclosure, the communication interaction between the client and the resource server MUST utilize confidentiality and integrity protection as described in Section 1.5.

There is no requirement on the particular structure or format of a bearer token, as described in Section 5. If a bearer token is a reference to authorization information, such references MUST be infeasible for an attacker to guess, such as using a sufficiently long cryptographically random string. If a bearer token uses an encoding mechanism to contain the authorization information in the token itself, the access token MUST use integrity protection sufficient to prevent the token from being modified. One example of an encoding and signing mechanism for access tokens is described in JSON Web Token Profile for Access Tokens [RFC9068].

### 5.2.1.  Authenticated Requests

This section defines two methods of sending Bearer tokens in resource requests to resource servers. Clients MUST use one of the two methods defined below, and MUST NOT use more than one method to transmit the token in each request.

In particular, clients MUST NOT send the access token in a URI query parameter, and resource servers MUST ignore access tokens in a URI query parameter.

### 5.2.1.1.  Authorization Request Header Field

When sending the access token in the Authorization request header field defined by HTTP/1.1 [RFC7235], the client uses the Bearer authentication scheme to transmit the access token.

For example:

```
GET /resource HTTP/1.1
Host: server.example.com
Authorization: Bearer mF_9.B5f-4.1JqM
```

The syntax of the Authorization header field for this scheme follows the usage of the Basic scheme defined in Section 2 of [RFC2617]. Note that, as with Basic, it does not conform to the generic syntax defined in Section 1.2 of [RFC2617] but is compatible with the general authentication framework in HTTP 1.1 Authentication [RFC7235], although it does not follow the preferred practice

outlined therein in order to reflect existing deployments. The
syntax for Bearer credentials is as follows:

```
b64token    = 1*( ALPHA / DIGIT /
                  "-" / "." / "_" / "~" / "+" / "/" ) *"="
credentials = "Bearer" 1*SP b64token
```

Clients SHOULD make authenticated requests with a bearer token using
the Authorization request header field with the Bearer HTTP
authorization scheme. Resource servers MUST support this method.

### 5.2.1.2.  Form-Encoded Content Parameter

When sending the access token in the HTTP request content, the
client adds the access token to the request content using the
access_token parameter. The client MUST NOT use this method unless
all of the following conditions are met:

  *The HTTP request includes the Content-Type header field set to
   application/x-www-form-urlencoded.

  *The content follows the encoding requirements of the application/
   x-www-form-urlencoded content-type as defined by the URL Living
   Standard [WHATWG.URL].

  *The HTTP request content is single-part.

  *The content to be encoded in the request MUST consist entirely of
   ASCII [USASCII] characters.

  *The HTTP request method is one for which the content has defined
   semantics. In particular, this means that the GET method MUST NOT
   be used.

The content MAY include other request-specific parameters, in which
case the access_token parameter MUST be properly separated from the
request-specific parameters using & character(s) (ASCII code 38).

For example, the client makes the following HTTP request using
transport-layer security:

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

access_token=mF_9.B5f-4.1JqM
```

The application/x-www-form-urlencoded method SHOULD NOT be used
except in application contexts where participating clients do not

have access to the Authorization request header field. Resource
servers MAY support this method.

### 5.2.2.  Access Token Validation

After receiving the access token, the resource server MUST check
that the access token is not yet expired, is authorized to access
the requested resource, was issued with the appropriate scope, and
meets other policy requirements of the resource server to access the
protected resource.

Access tokens generally fall into two categories: reference tokens
or self-encoded tokens. Reference tokens can be validated by
querying the authorization server or looking up the token in a token
database, whereas self-encoded tokens contain the authorization
information in an encrypted and/or signed string which can be
extracted by the resource server.

A standardized method to query the authorization server to check the
validity of an access token is defined in Token Introspection
([RFC7662]).

A standardized method of encoding information in a token string is
defined in JWT Profile for Access Tokens ([RFC9068]).

See Section 7.1 for additional considerations around creating and
validating access tokens.

### 5.2.3.  The WWW-Authenticate Response Header Field

If the protected resource request does not include authentication
credentials or does not contain an access token that enables access
to the protected resource, the resource server MUST include the HTTP
WWW-Authenticate response header field; it MAY include it in
response to other conditions as well. The WWW-Authenticate header
field uses the framework defined by HTTP/1.1 [RFC7235].

All challenges for this token type MUST use the auth-scheme value
Bearer. This scheme MUST be followed by one or more auth-param
values. The auth-param attributes used or defined by this
specification for this token type are as follows. Other auth-param
attributes MAY be used as well.

A realm attribute MAY be included to indicate the scope of
protection in the manner described in HTTP/1.1 [RFC7235]. The realm
attribute MUST NOT appear more than once.

The scope attribute is defined in Section 3.2.2.1. The scope
attribute is a space-delimited list of case-sensitive scope values
indicating the required scope of the access token for accessing the

requested resource. scope values are implementation defined; there
is no centralized registry for them; allowed values are defined by
the authorization server. The order of scope values is not
significant. In some cases, the scope value will be used when
requesting a new access token with sufficient scope of access to
utilize the protected resource. Use of the scope attribute is
OPTIONAL. The scope attribute MUST NOT appear more than once. The
scope value is intended for programmatic use and is not meant to be
displayed to end-users.

Two example scope values follow; these are taken from the OpenID
Connect [OpenID.Messages] and the Open Authentication Technology
Committee (OATC) Online Multimedia Authorization Protocol [OMAP]
OAuth 2.0 use cases, respectively:

```
scope="openid profile email"
scope="urn:example:channel=HBO&urn:example:rating=G,PG-13"
```

If the protected resource request included an access token and
failed authentication, the resource server SHOULD include the error
attribute to provide the client with the reason why the access
request was declined. The parameter value is described in
Section 5.2.4. In addition, the resource server MAY include the
error_description attribute to provide developers a human-readable
explanation that is not meant to be displayed to end-users. It also
MAY include the error_uri attribute with an absolute URI identifying
a human-readable web page explaining the error. The error,
error_description, and error_uri attributes MUST NOT appear more
than once.

Values for the scope attribute (specified in Appendix A.4) MUST NOT
include characters outside the set %x21 / %x23-5B / %x5D-7E for
representing scope values and %x20 for delimiters between scope
values. Values for the error and error_description attributes
(specified in Appendix A.7 and Appendix A.8) MUST NOT include
characters outside the set %x20-21 / %x23-5B / %x5D-7E. Values for
the error_uri attribute (specified in Appendix A.9 of) MUST conform
to the URI-reference syntax and thus MUST NOT include characters
outside the set %x21 / %x23-5B / %x5D-7E.

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

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example"
```

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

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

### 5.2.4.  Error Codes

When a request fails, the resource server responds using the
appropriate HTTP status code (typically, 400, 401, 403, or 405) and
includes one of the following error codes in the response:

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

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

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

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

For example:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example"
```

### 5.3.  Error Response

If a resource access request fails, the resource server SHOULD
inform the client of the error. The method by which the resource
server does this is determined by the particular token type, such as
the description of Bearer tokens in Section 5.2.4.

### 5.3.1. Extension Token Types

[RFC6750] establishes a common registry in Section 11.4 of [RFC6749] for error values to be shared among OAuth token authentication schemes.

New authentication schemes designed primarily for OAuth token authentication SHOULD define a mechanism for providing an error status code to the client, in which the error values allowed are registered in the error registry established by this specification.

Such schemes MAY limit the set of valid error codes to a subset of the registered values. If the error code is returned using a named parameter, the parameter name SHOULD be error.

Other schemes capable of being used for OAuth token authentication, but not primarily designed for that purpose, MAY bind their error values to the registry in the same manner.

New authentication schemes MAY choose to also specify the use of the error_description and error_uri parameters to return error information in a manner parallel to their usage in this specification.

## 6.  Extensibility

## 6.1.  Defining Access Token Types

Access token types can be defined in one of two ways: registered in the Access Token Types registry (following the procedures in Section 11.1 of [RFC6749]), 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
```

## 6.2.  Defining New Endpoint Parameters

New request or response parameters for use with the authorization endpoint or the token endpoint are defined and registered in the

OAuth Parameters registry following the procedure in Section 11.2 of
[RFC6749].

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 that 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_').

### 6.3.  Defining New Authorization Grant Types

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 of [RFC6749].

### 6.4.  Defining New Authorization Endpoint Response Types

New response types for use with the authorization endpoint are
defined and registered in the Authorization Endpoint Response Types
registry following the procedure in Section 11.3 of [RFC6749].
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, an extension can define and register the code
other_token response type. Once registered, the same combination
cannot be registered as other_token code, but both values can be
used to denote the same response type.

### 6.5.  Defining Additional Error Codes

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 token error response
(Section 3.2.3.1), or the resource access error response
(Section 5.3), such error codes MAY be defined.

Extension error codes MUST be registered (following the procedures
in Section 11.4 of [RFC6749]) 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 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       = 1*error-char
error-char = %x20-21 / %x23-5B / %x5D-7E
```

## 7.  Security Considerations

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, browser-
based application, and native application.

A comprehensive OAuth security model and analysis, as well as
background for the protocol design, is provided by [RFC6819] and
[I-D.ietf-oauth-security-topics].

### 7.1.  Access Token Security Considerations

### 7.1.1.  Security Threats

The following list presents several common threats against protocols
utilizing some form of tokens. This list of threats is based on NIST
Special Publication 800-63 [NIST800-63].

#### 7.1.1.1.  Access token manufacture/modification

An attacker may generate a bogus access token or modify the token
contents (such as the authentication or attribute statements) of an
existing token, causing the resource server to grant inappropriate
access to the client. For example, an attacker may modify the token
to extend the validity period; a malicious client may modify the
assertion to gain access to information that they should not be able
to view.

### 7.1.1.2.  Access token disclosure

Access tokens may contain authentication and attribute statements that include sensitive information.

### 7.1.1.3.  Access token redirect

An attacker uses an access token generated for consumption by one resource server to gain access to a different resource server that mistakenly believes the token to be for it.

### 7.1.1.4.  Access token replay

An attacker attempts to use an access token that has already been used with that resource server in the past.

### 7.1.2.  Threat Mitigation

A large range of threats can be mitigated by protecting the contents of the access token by using a digital signature.

Alternatively, a bearer token can contain a reference to authorization information, rather than encoding the information directly. Using a reference may require an extra interaction between a resource server and authorization server to resolve the reference to the authorization information. The mechanics of such an interaction are not defined by this specification, but one such mechanism is defined in Token Introspection [RFC7662].

This document does not specify the encoding or the contents of the access token; hence, detailed recommendations about the means of guaranteeing access token integrity protection are outside the scope of this specification. One example of an encoding and signing mechanism for access tokens is described in JSON Web Token Profile for Access Tokens [RFC9068].

To deal with access token redirects, it is important for the authorization server to include the identity of the intended recipients (the audience), typically a single resource server (or a list of resource servers), in the token. Restricting the use of the token to a specific scope is also RECOMMENDED.

If cookies are transmitted without TLS protection, any information contained in them is at risk of disclosure. Therefore, Bearer tokens MUST NOT be stored in cookies that can be sent in the clear, as any information in them is at risk of disclosure. See "HTTP State Management Mechanism" [RFC6265] for security considerations about cookies.

In some deployments, including those utilizing load balancers, the TLS connection to the resource server terminates prior to the actual server that provides the resource. This could leave the token unprotected between the front-end server where the TLS connection terminates and the back-end server that provides the resource. In such deployments, sufficient measures MUST be employed to ensure confidentiality of the access token between the front-end and back-end servers; encryption of the token is one such possible measure.

### 7.1.3.  Summary of Recommendations

### 7.1.3.1.  Safeguard bearer tokens

Client implementations MUST ensure that bearer tokens are not leaked to unintended parties, as they will be able to use them to gain access to protected resources. This is the primary security consideration when using bearer tokens and underlies all the more specific recommendations that follow.

### 7.1.3.2.  Validate TLS certificate chains

The client MUST validate the TLS certificate chain when making requests to protected resources. Failing to do so may enable DNS hijacking attacks to steal the token and gain unintended access.

### 7.1.3.3.  Always use TLS (https)

Clients MUST always use TLS (https) or equivalent transport security when making requests with bearer tokens. Failing to do so exposes the token to numerous attacks that could give attackers unintended access.

### 7.1.3.4.  Don't store bearer tokens in HTTP cookies

Implementations MUST NOT store bearer tokens within cookies that can be sent in the clear (which is the default transmission mode for cookies). Implementations that do store bearer tokens in cookies MUST take precautions against cross-site request forgery.

### 7.1.3.5.  Issue short-lived bearer tokens

Authorization servers SHOULD issue short-lived bearer tokens, particularly when issuing tokens to clients that run within a web browser or other environments where information leakage may occur. Using short-lived bearer tokens can reduce the impact of them being leaked.

### 7.1.3.6.  Issue scoped bearer tokens

Authorization servers SHOULD issue bearer tokens that contain an
audience restriction, scoping their use to the intended relying
party or set of relying parties.

### 7.1.3.7.  Don't pass bearer tokens in page URLs

Bearer tokens MUST NOT be passed in page URLs (for example, as query
string parameters). Instead, bearer tokens SHOULD be passed in HTTP
message headers or message bodies for which confidentiality measures
are taken. Browsers, web servers, and other software may not
adequately secure URLs in the browser history, web server logs, and
other data structures. If bearer tokens are passed in page URLs,
attackers might be able to steal them from the history data, logs,
or other unsecured locations.

### 7.1.4.  Sender-Constrained Access Tokens

A sender-constrained access token scopes the applicability of an
access token to a certain sender. This sender is obliged to
demonstrate knowledge of a certain secret as prerequisite for the
acceptance of that access token at the recipient (e.g., a resource
server).

Authorization and resource servers SHOULD use mechanisms for sender-
constrained access tokens to prevent token replay as described in
Section 4.8.1.1.2 of [I-D.ietf-oauth-security-topics]. The use of
Mutual TLS for OAuth 2.0 [RFC8705] is RECOMMENDED as the method of
sender-constrained access tokens. Another method of achieving
sender-constrained tokens is described in [I-D.ietf-oauth-dpop].

It is RECOMMENDED to use end-to-end TLS. If TLS traffic needs to be
terminated at an intermediary, refer to Section 4.11 of
[I-D.ietf-oauth-security-topics] for further security advice.

### 7.1.5.  Access Token Privilege Restriction

The privileges associated with an access token SHOULD be restricted
to the minimum required for the particular application or use case.
This prevents clients from exceeding the privileges authorized by
the resource owner. It also prevents users from exceeding their
privileges authorized by the respective security policy. Privilege
restrictions also help to reduce the impact of access token leakage.

In particular, access tokens SHOULD be restricted to certain
resource servers (audience restriction), preferably to a single
resource server. To put this into effect, the authorization server
associates the access token with certain resource servers and every
resource server is obliged to verify, for every request, whether the

access token sent with that request was meant to be used for that particular resource server. If not, the resource server MUST refuse to serve the respective request. Clients and authorization servers MAY utilize the parameters scope or resource as specified in this document and [RFC8707], respectively, to determine the resource server they want to access.

Additionally, access tokens SHOULD be restricted to certain resources and actions on resource servers or resources. To put this into effect, the authorization server associates the access token with the respective resource and actions and every resource server is obliged to verify, for every request, whether the access token sent with that request was meant to be used for that particular action on the particular resource. If not, the resource server must refuse to serve the respective request. Clients and authorization servers MAY utilize the parameter scope and authorization_details as specified in [I-D.ietf-oauth-rar] to determine those resources and/ or actions.

## 7.2.  Client Authentication

Depending on the overall process of client registration and credential lifecycle management, this may affect the confidence an authorization server has in a particular client.

For example, authentication of a dynamically registered client does not prove the identity of the client, it only ensures that repeated requests to the authorization server were made from the same client instance. Such clients may be limited in terms of which scopes they are allowed to request, or may have other limitations such as shorter token lifetimes.

In contrast, if there is a registered application whose developer's identity was verified, who signed a contract and is issued a client secret that is only used in a secure backend service, the authorization server might allow this client to request more sensitive scopes or to be issued longer-lasting tokens.

## 7.3.  Client Impersonation

If a confidential client has its credentials stolen, a malicious client can impersonate the client and obtain access to protected resources.

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 to 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 that the repeated request comes from the original client and
not an impersonator.

### 7.3.1.  Impersonation of Native Apps

As stated above, the authorization server SHOULD NOT process
authorization requests automatically without user consent or
interaction, except when the identity of the client can be assured.
This includes the case where the user has previously approved an
authorization request for a given client ID -- unless the identity
of the client can be proven, the request SHOULD be processed as if
no previous request had been approved.

Measures such as claimed https scheme redirects MAY be accepted by
authorization servers as identity proof. Some operating systems may
offer alternative platform-specific identity features that MAY be
accepted, as appropriate.

### 7.3.2.  Access Token Privilege Restriction

The client SHOULD request access tokens with the minimal scope
necessary. The authorization server SHOULD take the client identity
into account when choosing how to honor the requested scope and MAY
issue an access token with fewer scopes than requested.

The privileges associated with an access token SHOULD be restricted
to the minimum required for the particular application or use case.
This prevents clients from exceeding the privileges authorized by
the resource owner. It also prevents users from exceeding their
privileges authorized by the respective security policy. Privilege
restrictions also help to reduce the impact of access token leakage.

In particular, access tokens SHOULD be restricted to certain
resource servers (audience restriction), preferably to a single
resource server. To put this into effect, the authorization server
associates the access token with certain resource servers and every
resource server is obliged to verify, for every request, whether the
access token sent with that request was meant to be used for that
particular resource server. If not, the resource server MUST refuse
to serve the respective request. Clients and authorization servers
MAY utilize the parameters scope or resource as specified in
[RFC8707], respectively, to determine the resource server they want
to access.

### 7.4. Client Impersonating Resource Owner

Resource servers may make access control decisions based on the identity of a resource owner, for which an access token was issued, or based on the identity of a client in the client credentials grant. If both options are possible, depending on the details of the implementation, a client's identity may be mistaken for the identity of a resource owner. For example, if a client is able to choose its own client_id during registration with the authorization server, a malicious client may set it to a value identifying an end-user (e.g., a sub value if OpenID Connect is used). If the resource server cannot properly distinguish between access tokens issued to clients and access tokens issued to end-users, the client may then be able to access resource of the end-user.

Authorization servers SHOULD NOT allow clients to influence their client_id or any other Claim if that can cause confusion with a genuine resource owner. Where this cannot be avoided, authorization servers MUST provide other means for the resource server to distinguish between access tokens authorized by a resource owner from access tokens authorized by the client itself.

### 7.5. Protecting the Authorization Code Flow

### 7.5.1. Loopback Redirect Considerations in Native Apps

Loopback interface redirect URIs MAY use the http scheme (i.e., without TLS). This is acceptable for loopback interface redirect URIs as the HTTP request never leaves the device.

Clients should open the network port only when starting the authorization request and close it once the response is returned.

Clients should listen on the loopback network interface only, in order to avoid interference by other network actors.

Clients should use loopback IP literals rather than the string localhost as described in Section 8.4.2.

### 7.5.2. HTTP 307 Redirect

An AS which redirects a request that potentially contains user credentials MUST NOT use the 307 status code (see Section 15.4.8 of [RFC9110]) for redirection. If an HTTP redirection (and not, for example, JavaScript) is used for such a request, AS SHOULD use the status code 303 "See Other".

At the authorization endpoint, a typical protocol flow is that the AS prompts the user to enter their credentials in a form that is then submitted (using the POST method) back to the authorization

server. The AS checks the credentials and, if successful, redirects the user agent to the client's redirect URI.

If the status code 307 were used for redirection, the user agent would send the user credentials via a POST request to the client.

This discloses the sensitive credentials to the client. If the relying party is malicious, it can use the credentials to impersonate the user at the AS.

The behavior might be unexpected for developers, but is defined in Section 15.4.8 of [RFC9110]. This status code does not require the user agent to rewrite the POST request to a GET request and thereby drop the form data in the POST request content.

In HTTP [RFC9110], only the status code 303 unambigiously enforces rewriting the HTTP POST request to an HTTP GET request. For all other status codes, including the popular 302, user agents can opt not to rewrite POST to GET requests and therefore to reveal the user credentials to the client. (In practice, however, most user agents will only show this behaviour for 307 redirects.)

Therefore, the RECOMMENDED status code for HTTP redirects is 303.

## 7.6.  Authorization Code Injection

TODO: Expand this with background on what authorization code injection is

### 7.6.1.  Countermeasures

To prevent injection of authorization codes into the client, using code_challenge and code_verifier is REQUIRED for clients, and authorization servers MUST enforce their use, unless both of the following criteria are met:

  *The client is a confidential client.

  *In the specific deployment and the specific request, there is reasonable assurance by the authorization server that the client implements the OpenID Connect nonce mechanism properly.

In this case, using and enforcing code_challenge and code_verifier is still RECOMMENDED.

The code_challenge or OpenID Connect nonce value MUST be transaction-specific and securely bound to the client and the user agent in which the transaction was started. If a transaction leads to an error, fresh values for code_challenge or nonce MUST be chosen.

Historic note: Although PKCE [RFC7636] (where the code_challenge and code_verifier parameters were created) was originally designed as a mechanism to protect native apps, this advice applies to all kinds of OAuth clients, including web applications and other confidential clients.

## 7.7.  Ensuring Endpoint Authenticity

The risk related to man-in-the-middle attacks is mitigated by the mandatory use of channel security mechanisms such as [RFC8446] for communicating with the Authorization and Token Endpoints. See Section 1.5 for further details.

## 7.8.  Credentials-Guessing Attacks

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

The probability of an attacker guessing generated tokens (and other credentials not intended for handling by end-users) 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.

## 7.9.  Phishing Attacks

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.

See Section 1.5 for further details on mitigating the risk of phishing attacks.

## 7.10.  Cross-Site Request Forgery

An attacker might attempt to inject a request to the redirect URI of the legitimate client on the victim's device, e.g., to cause the client to access resources under the attacker's control. This is a variant of an attack known as Cross-Site Request Forgery (CSRF).

The traditional countermeasure are CSRF tokens that are bound to the user agent and passed in the state parameter to the authorization server as described in [RFC6819]. The same protection is provided by the code_verifier parameter or the OpenID Connect nonce value.

When using code_verifier instead of state or nonce for CSRF protection, it is important to note that:

  *Clients MUST ensure that the AS supports the
   code_challenge_method intended to be used by the client. If an
   authorization server does not support the requested method, state
   or nonce MUST be used for CSRF protection instead.

  *If state is used for carrying application state, and integrity of
   its contents is a concern, clients MUST protect state against
   tampering and swapping. This can be achieved by binding the
   contents of state to the browser session and/or signed/encrypted
   state values [I-D.bradley-oauth-jwt-encoded-state].

AS therefore MUST provide a way to detect their supported code challenge methods either via AS metadata according to [RFC8414] or provide a deployment-specific way to ensure or determine support.

## 7.11.  Clickjacking

As described in Section 4.4.1.9 of [RFC6819], the authorization request is susceptible to clickjacking attacks, also called user interface redressing. In such an attack, an attacker embeds the authorization endpoint user interface in an innocuous context. A user believing to interact with that context, for example, clicking on buttons, inadvertently interacts with the authorization endpoint user interface instead. The opposite can be achieved as well: A user believing to interact with the authorization endpoint might inadvertently type a password into an attacker-provided input field overlaid over the original user interface. Clickjacking attacks can be designed such that users can hardly notice the attack, for example using almost invisible iframes overlaid on top of other elements.

An attacker can use this vector to obtain the user's authentication credentials, change the scope of access granted to the client, and potentially access the user's resources.

Authorization servers MUST prevent clickjacking attacks. Multiple countermeasures are described in [RFC6819], including the use of the X-Frame-Options HTTP response header field and frame-busting JavaScript. In addition to those, authorization servers SHOULD also use Content Security Policy (CSP) level 2 [CSP-2] or greater.

To be effective, CSP must be used on the authorization endpoint and, if applicable, other endpoints used to authenticate the user and authorize the client (e.g., the device authorization endpoint, login pages, error pages, etc.). This prevents framing by unauthorized origins in user agents that support CSP. The client MAY permit being framed by some other origin than the one used in its redirection endpoint. For this reason, authorization servers SHOULD allow administrators to configure allowed origins for particular clients and/or for clients to register these dynamically.

Using CSP allows authorization servers to specify multiple origins in a single response header field and to constrain these using flexible patterns (see [CSP-2] for details). Level 2 of this standard provides a robust mechanism for protecting against clickjacking by using policies that restrict the origin of frames (using frame-ancestors) together with those that restrict the sources of scripts allowed to execute on an HTML page (by using script-src). A non-normative example of such a policy is shown in the following listing:

HTTP/1.1 200 OK Content-Security-Policy: frame-ancestors https:// ext.example.org:8000 Content-Security-Policy: script-src 'self' X-Frame-Options: ALLOW-FROM https://ext.example.org:8000 ...

Because some user agents do not support [CSP-2], this technique SHOULD be combined with others, including those described in [RFC6819], unless such legacy user agents are explicitly unsupported by the authorization server. Even in such cases, additional countermeasures SHOULD still be employed.

## 7.12.  Code Injection and Input Validation

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 introduce a wide range of malicious side-effects.

The authorization server and client MUST sanitize (and validate when possible) any value received -- in particular, the value of the state and redirect_uri parameters.

### 7.13.  Open Redirection

The following attacks can occur when an AS or client has an open
redirector. An open redirector is an endpoint that forwards a user's
browser to an arbitrary URI obtained from a query parameter. Such
endpoints are sometimes implemented, for example, to show a message
before a user is then redirected to an external website, or to
redirect users back to a URL they were intending to visit before
being interrupted, e.g., by a login prompt.

### 7.13.1.  Client as Open Redirector

Clients MUST NOT expose open redirectors. Attackers may use open
redirectors to produce URLs pointing to the client and utilize them
to exfiltrate authorization codes, as described in Section 4.1.1 of
[I-D.ietf-oauth-security-topics]. Another abuse case is to produce
URLs that appear to point to the client. This might trick users into
trusting the URL and follow it in their browser. This can be abused
for phishing.

In order to prevent open redirection, clients should only redirect
if the target URLs are whitelisted or if the origin and integrity of
a request can be authenticated. Countermeasures against open
redirection are described by OWASP [owasp_redir].

### 7.13.2.  Authorization Server as Open Redirector

Just as with clients, attackers could try to utilize a user's trust
in the authorization server (and its URL in particular) for
performing phishing attacks. OAuth authorization servers regularly
redirect users to other web sites (the clients), but must do so in a
safe way.

Section 4.1.2.1 already prevents open redirects by stating that the
AS MUST NOT automatically redirect the user agent in case of an
invalid combination of client_id and redirect_uri.

However, an attacker could also utilize a correctly registered
redirect URI to perform phishing attacks. The attacker could, for
example, register a client via dynamic client registration [RFC7591]
and execute one of the following attacks:

1.  Intentionally send an erroneous authorization request, e.g., by
    using an invalid scope value, thus instructing the AS to
    redirect the user-agent to its phishing site.

2.  Intentionally send a valid authorization request with client_id
    and redirect_uri controlled by the attacker. After the user
    authenticates, the AS prompts the user to provide consent to
    the request. If the user notices an issue with the request and

declines the request, the AS still redirects the user agent to
the phishing site. In this case, the user agent will be
redirected to the phishing site regardless of the action taken
by the user.

3. Intentionally send a valid silent authentication request
   (prompt=none) with client_id and redirect_uri controlled by the
   attacker. In this case, the AS will automatically redirect the
   user agent to the phishing site.

The AS MUST take precautions to prevent these threats. The AS MUST
always authenticate the user first and, with the exception of the
silent authentication use case, prompt the user for credentials when
needed, before redirecting the user. Based on its risk assessment,
the AS needs to decide whether it can trust the redirect URI or not.
It could take into account URI analytics done internally or through
some external service to evaluate the credibility and
trustworthiness content behind the URI, and the source of the
redirect URI and other client data.

The AS SHOULD only automatically redirect the user agent if it
trusts the redirect URI. If the URI is not trusted, the AS MAY
inform the user and rely on the user to make the correct decision.

## 7.14.  Authorization Server Mix-Up Mitigation in Native Apps

(TODO: merge this with the regular mix-up section when it is brought
in)

To protect against a compromised or malicious authorization server
attacking another authorization server used by the same app, it is
REQUIRED that a unique redirect URI is used for each authorization
server used by the app (for example, by varying the path component),
and that authorization responses are rejected if the redirect URI
they were received on doesn't match the redirect URI in an outgoing
authorization request.

The native app MUST store the redirect URI used in the authorization
request with the authorization session data (i.e., along with state
and other related data) and MUST verify that the URI on which the
authorization response was received exactly matches it.

The requirement of [Section 8.1](), specifically that authorization
servers reject requests with URIs that don't match what was
registered, is also required to prevent such attacks.

## 8.  Native Applications

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. The best current practice is to perform the OAuth authorization request in an external user agent (typically the browser) rather than an embedded user agent (such as one implemented with web-views).

The native application can capture the response from the authorization server using a redirect 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 redirect URI identifying a server-hosted resource under the client's control, which in turn makes the response available to the native application.

Previously, it was common for native apps to use embedded user agents (commonly implemented with web-views) for OAuth authorization requests. That approach has many drawbacks, including the host app being able to copy user credentials and cookies as well as the user needing to authenticate from scratch in each app. See Section 8.5.1 for a deeper analysis of the drawbacks of using embedded user agents for OAuth.

Native app authorization requests that use the system browser are more secure and can take advantage of the user's authentication state on the device. Being able to use the existing authentication session in the browser enables single sign-on, as users don't need to authenticate to the authorization server each time they use a new app (unless required by the authorization server policy).

Supporting authorization flows between a native app and the browser is possible without changing the OAuth protocol itself, as the OAuth authorization request and response are already defined in terms of URIs. This encompasses URIs that can be used for inter-app communication. Some OAuth server implementations that assume all clients are confidential web clients will need to add an understanding of public native app clients and the types of redirect URIs they use to support this best practice.

## 8.1.  Registration of Native App Clients

Except when using a mechanism like Dynamic Client Registration [RFC7591] to provision per-instance secrets, native apps are classified as public clients, as defined in Section 2.1; they MUST be registered with the authorization server as such. Authorization

servers MUST record the client type in the client registration
details in order to identify and process requests accordingly.

### 8.1.1.  Client Authentication of Native Apps

Secrets that are statically included as part of an app distributed
to multiple users should not be treated as confidential secrets, as
one user may inspect their copy and learn the shared secret. For
this reason, it is NOT RECOMMENDED for authorization servers to
require client authentication of public native apps clients using a
shared secret, as this serves little value beyond client
identification which is already provided by the client_id request
parameter.

Authorization servers that still require a statically included
shared secret for native app clients MUST treat the client as a
public client (as defined in Section 2.1), and not accept the secret
as proof of the client's identity. Without additional measures, such
clients are subject to client impersonation (see Section 7.3.1).

### 8.2.  Using Inter-App URI Communication for OAuth in Native Apps

Just as URIs are used for OAuth on the web to initiate the
authorization request and return the authorization response to the
requesting website, URIs can be used by native apps to initiate the
authorization request in the device's browser and return the
response to the requesting native app.

By adopting the same methods used on the web for OAuth, benefits
seen in the web context like the usability of a single sign-on
session and the security of a separate authentication context are
likewise gained in the native app context. Reusing the same approach
also reduces the implementation complexity and increases
interoperability by relying on standards-based web flows that are
not specific to a particular platform.

Native apps MUST use an external user agent to perform OAuth
authorization requests. This is achieved by opening the
authorization request in the browser (detailed in Section 8.3) and
using a redirect URI that will return the authorization response
back to the native app (defined in Section 8.4).

### 8.3.  Initiating the Authorization Request from a Native App

Native apps needing user authorization create an authorization
request URI with the authorization code grant type per Section 4.1
using a redirect URI capable of being received by the native app.

The function of the redirect URI for a native app authorization
request is similar to that of a web-based authorization request.

Rather than returning the authorization response to the OAuth client's server, the redirect URI used by a native app returns the response to the app. Several options for a redirect URI that will return the authorization response to the native app in different platforms are documented in [Section 8.4](#). Any redirect URI that allows the app to receive the URI and inspect its parameters is viable.

After constructing the authorization request URI, the app uses platform-specific APIs to open the URI in an external user agent. Typically, the external user agent used is the default browser, that is, the application configured for handling http and https scheme URIs on the system; however, different browser selection criteria and other categories of external user agents MAY be used.

This best practice focuses on the browser as the RECOMMENDED external user agent for native apps. An external user agent designed specifically for user authorization and capable of processing authorization requests and responses like a browser MAY also be used. Other external user agents, such as a native app provided by the authorization server may meet the criteria set out in this best practice, including using the same redirect URI properties, but their use is out of scope for this specification.

Some platforms support a browser feature known as "in-app browser tabs", where an app can present a tab of the browser within the app context without switching apps, but still retain key benefits of the browser such as a shared authentication state and security context. On platforms where they are supported, it is RECOMMENDED, for usability reasons, that apps use in-app browser tabs for the authorization request.

## 8.4. Receiving the Authorization Response in a Native App

There are several redirect URI options available to native apps for receiving the authorization response from the browser, the availability and user experience of which varies by platform.

### 8.4.1. Claimed "https" Scheme URI Redirection

Some operating systems allow apps to claim https URIs (see Section 4.2.2 of [[RFC9110](#)]) in the domains they control. When the browser encounters a claimed URI, instead of the page being loaded in the browser, the native app is launched with the URI supplied as a launch parameter.

Such URIs can be used as redirect URIs by native apps. They are indistinguishable to the authorization server from a regular web-based client redirect URI. An example is:

```
https://app.example.com/oauth2redirect/example-provider
```

   As the redirect URI alone is not enough to distinguish public native
   app clients from confidential web clients, it is REQUIRED in
   [Section 8.1](#) that the client type be recorded during client
   registration to enable the server to determine the client type and
   act accordingly.

   App-claimed https scheme redirect URIs have some advantages compared
   to other native app redirect options in that the identity of the
   destination app is guaranteed to the authorization server by the
   operating system. For this reason, native apps SHOULD use them over
   the other options where possible.

### 8.4.2.  Loopback Interface Redirection

   Native apps that are able to open a port on the loopback network
   interface without needing special permissions (typically, those on
   desktop operating systems) can use the loopback interface to receive
   the OAuth redirect.

   Loopback redirect URIs use the http scheme and are constructed with
   the loopback IP literal and whatever port the client is listening
   on.

   That is, http://127.0.0.1:{port}/{path} for IPv4, and http://[::1]:
   {port}/{path} for IPv6. An example redirect using the IPv4 loopback
   interface with a randomly assigned port:

```
http://127.0.0.1:51004/oauth2redirect/example-provider
```

   An example redirect using the IPv6 loopback interface with a
   randomly assigned port:

```
http://[::1]:61023/oauth2redirect/example-provider
```

   While redirect URIs using the name localhost (i.e., http://
   localhost:{port}/{path}) function similarly to loopback IP
   redirects, the use of localhost is NOT RECOMMENDED. Specifying a
   redirect URI with the loopback IP literal rather than localhost
   avoids inadvertently listening on network interfaces other than the
   loopback interface. It is also less susceptible to client-side
   firewalls and misconfigured host name resolution on the user's
   device.

   The authorization server MUST allow any port to be specified at the
   time of the request for loopback IP redirect URIs, to accommodate
   clients that obtain an available ephemeral port from the operating
   system at the time of the request.

Clients SHOULD NOT assume that the device supports a particular
version of the Internet Protocol. It is RECOMMENDED that clients
attempt to bind to the loopback interface using both IPv4 and IPv6
and use whichever is available.

### 8.4.3.  Private-Use URI Scheme Redirection

Many mobile and desktop computing platforms support inter-app
communication via URIs by allowing apps to register private-use URI
schemes (sometimes colloquially referred to as "custom URL schemes")
like com.example.app. When the browser or another app attempts to
load a URI with a private-use URI scheme, the app that registered it
is launched to handle the request.

Many environments that support private-use URI schemes do not
provide a mechanism to claim a scheme and prevent other parties from
using another application's scheme. As such, clients using private-
use URI schemes are vulnerable to potential attacks on their
redirect URIs, so this option should only be used if the previously
mentioned more secure options are not available.

To perform an authorization request with a private-use URI scheme
redirect, the native app launches the browser with a standard
authorization request, but one where the redirect URI utilizes a
private-use URI scheme it registered with the operating system.

When choosing a URI scheme to associate with the app, apps MUST use
a URI scheme based on a domain name under their control, expressed
in reverse order, as recommended by Section 3.8 of [RFC7595] for
private-use URI schemes.

For example, an app that controls the domain name app.example.com
can use com.example.app as their scheme. Some authorization servers
assign client identifiers based on domain names, for example,
client1234.usercontent.example.net, which can also be used as the
domain name for the scheme when reversed in the same manner. A
scheme such as myapp, however, would not meet this requirement, as
it is not based on a domain name.

When there are multiple apps by the same publisher, care must be
taken so that each scheme is unique within that group. On platforms
that use app identifiers based on reverse-order domain names, those
identifiers can be reused as the private-use URI scheme for the
OAuth redirect to help avoid this problem.

Following the requirements of Section 3.2 of [RFC3986], as there is
no naming authority for private-use URI scheme redirects, only a
single slash (/) appears after the scheme component. A complete
example of a redirect URI utilizing a private-use URI scheme is:

```
com.example.app:/oauth2redirect/example-provider
```

When the authorization server completes the request, it redirects to
the client's redirect URI as it would normally. As the redirect URI
uses a private-use URI scheme, it results in the operating system
launching the native app, passing in the URI as a launch parameter.
Then, the native app uses normal processing for the authorization
response.

## 8.5.  Security Considerations in Native Apps

### 8.5.1.  Embedded User Agents in Native Apps

Embedded user agents are a technically possible method for
authorizing native apps. These embedded user agents are unsafe for
use by third parties to the authorization server by definition, as
the app that hosts the embedded user agent can access the user's
full authentication credentials, not just the OAuth authorization
grant that was intended for the app.

In typical web-view-based implementations of embedded user agents,
the host application can record every keystroke entered in the login
form to capture usernames and passwords, automatically submit forms
to bypass user consent, and copy session cookies and use them to
perform authenticated actions as the user.

Even when used by trusted apps belonging to the same party as the
authorization server, embedded user agents violate the principle of
least privilege by having access to more powerful credentials than
they need, potentially increasing the attack surface.

Encouraging users to enter credentials in an embedded user agent
without the usual address bar and visible certificate validation
features that browsers have makes it impossible for the user to know
if they are signing in to the legitimate site; even when they are,
it trains them that it's OK to enter credentials without validating
the site first.

Aside from the security concerns, embedded user agents do not share
the authentication state with other apps or the browser, requiring
the user to log in for every authorization request, which is often
considered an inferior user experience.

### 8.5.2.  Fake External User-Agents in Native Apps

The native app that is initiating the authorization request has a
large degree of control over the user interface and can potentially
present a fake external user agent, that is, an embedded user agent
made to appear as an external user agent.

When all good actors are using external user agents, the advantage is that it is possible for security experts to detect bad actors, as anyone faking an external user agent is provably bad. On the other hand, if good and bad actors alike are using embedded user agents, bad actors don't need to fake anything, making them harder to detect. Once a malicious app is detected, it may be possible to use this knowledge to blacklist the app's signature in malware scanning software, take removal action (in the case of apps distributed by app stores) and other steps to reduce the impact and spread of the malicious app.

Authorization servers can also directly protect against fake external user agents by requiring an authentication factor only available to true external user agents.

Users who are particularly concerned about their security when using in-app browser tabs may also take the additional step of opening the request in the full browser from the in-app browser tab and complete the authorization there, as most implementations of the in-app browser tab pattern offer such functionality.

### 8.5.3. Malicious External User-Agents in Native Apps

If a malicious app is able to configure itself as the default handler for https scheme URIs in the operating system, it will be able to intercept authorization requests that use the default browser and abuse this position of trust for malicious ends such as phishing the user.

This attack is not confined to OAuth; a malicious app configured in this way would present a general and ongoing risk to the user beyond OAuth usage by native apps. Many operating systems mitigate this issue by requiring an explicit user action to change the default handler for http and https scheme URIs.

### 9. Browser-Based Apps

Browser-based apps are are clients that run in a web browser, typically written in JavaScript, also known as "single-page apps". These types of apps have particular security considerations similar to native apps.

TODO: Bring in the normative text of the browser-based apps BCP when it is finalized.

### 10. Differences from OAuth 2.0

This draft consolidates the functionality in OAuth 2.0 [RFC6749], OAuth 2.0 for Native Apps ([RFC8252]), Proof Key for Code Exchange ([RFC7636]), OAuth 2.0 for Browser-Based Apps

([I-D.ietf-oauth-browser-based-apps]), OAuth Security Best Current
Practice ([I-D.ietf-oauth-security-topics]), and Bearer Token Usage
([RFC6750]).

Where a later draft updates or obsoletes functionality found in the
original [RFC6749], that functionality in this draft is updated with
the normative changes described in a later draft, or removed
entirely.

A non-normative list of changes from OAuth 2.0 is listed below:

  *The authorization code grant is extended with the functionality
   from PKCE ([RFC7636]) such that the default method of using the
   authorization code grant according to this specification requires
   the addition of the PKCE parameters

  *Redirect URIs must be compared using exact string matching as per
   Section 4.1.3 of [I-D.ietf-oauth-security-topics]

  *The Implicit grant (response_type=token) is omitted from this
   specification as per Section 2.1.2 of
   [I-D.ietf-oauth-security-topics]

  *The Resource Owner Password Credentials grant is omitted from
   this specification as per Section 2.4 of
   [I-D.ietf-oauth-security-topics]

  *Bearer token usage omits the use of bearer tokens in the query
   string of URIs as per Section 4.3.2 of
   [I-D.ietf-oauth-security-topics]

  *Refresh tokens for public clients must either be sender-
   constrained or one-time use as per Section 4.13.2 of
   [I-D.ietf-oauth-security-topics]

## 10.1.  Removal of the OAuth 2.0 Implicit grant

The OAuth 2.0 Implicit grant is omitted from OAuth 2.1 as it was
deprecated in [I-D.ietf-oauth-security-topics].

The intent of removing the Implicit grant is to no longer issue
access tokens in the authorization response, as such tokens are
vulnerable to leakage and injection, and are unable to be sender-
constrained to a client. This behavior was indicated by clients
using the response_type=token parameter. This value for the
response_type parameter is no longer defined in OAuth 2.1.

Removal of response_type=token does not have an effect on other
extension response types returning other artifacts from the

authorization endpoint, for example, response_type=id_token defined
by [OpenID].

## 11. IANA Considerations

This document does not require any IANA actions.

All referenced registries are defined by [RFC6749] and related
documents that this work is based upon. No changes to those
registries are required by this specification.

## 12. References

### 12.1. Normative References

[BCP195]    Saint-Andre, P., "Recommendations for Secure Use of
            Transport Layer Security (TLS)", 2015.

[I-D.ietf-oauth-security-topics] Lodderstedt, T., Bradley, J.,
            Labunets, A., and D. Fett, "OAuth 2.0 Security Best
            Current Practice", Work in Progress, Internet-Draft,
            draft-ietf-oauth-security-topics-22, 13 March 2023,
            <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-
            security-topics-22>.

[RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
            Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/
            RFC2119, March 1997, <https://www.rfc-editor.org/info/
            rfc2119>.

[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, DOI 10.17487/RFC2617, June 1999, <https://
            www.rfc-editor.org/info/rfc2617>.

[RFC3986]   Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform
            Resource Identifier (URI): Generic Syntax", STD 66, RFC
            3986, DOI 10.17487/RFC3986, January 2005, <https://
            www.rfc-editor.org/info/rfc3986>.

[RFC4949]   Shirey, R., "Internet Security Glossary, Version 2", FYI
            36, RFC 4949, DOI 10.17487/RFC4949, August 2007,
            <https://www.rfc-editor.org/info/rfc4949>.

[RFC5234]   Crocker, D., Ed. and P. Overell, "Augmented BNF for
            Syntax Specifications: ABNF", STD 68, RFC 5234, DOI
            10.17487/RFC5234, January 2008, <https://www.rfc-
            editor.org/info/rfc5234>.

[RFC6749]   Hardt, D., Ed., "The OAuth 2.0 Authorization Framework",
            RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://
            www.rfc-editor.org/info/rfc6749>.

[RFC6750]   Jones, M. and D. Hardt, "The OAuth 2.0 Authorization
            Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/
            RFC6750, October 2012, <https://www.rfc-editor.org/info/
            rfc6750>.

[RFC7159]   Bray, T., Ed., "The JavaScript Object Notation (JSON)
            Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159,
            March 2014, <https://www.rfc-editor.org/info/rfc7159>.

[RFC7235]   Fielding, R., Ed. and J. Reschke, Ed., "Hypertext
            Transfer Protocol (HTTP/1.1): Authentication", RFC 7235,
            DOI 10.17487/RFC7235, June 2014, <https://www.rfc-
            editor.org/info/rfc7235>.

[RFC7523]   Jones, M., Campbell, B., and C. Mortimore, "JSON Web
            Token (JWT) Profile for OAuth 2.0 Client Authentication
            and Authorization Grants", RFC 7523, DOI 10.17487/
            RFC7523, May 2015, <https://www.rfc-editor.org/info/
            rfc7523>.

[RFC7595]   Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines
            and Registration Procedures for URI Schemes", BCP 35, RFC
            7595, DOI 10.17487/RFC7595, June 2015, <https://www.rfc-
            editor.org/info/rfc7595>.

[RFC8174]   Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC
            2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174,
            May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8252]   Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps",
            BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017,
            <https://www.rfc-editor.org/info/rfc8252>.

[RFC8259]   Bray, T., Ed., "The JavaScript Object Notation (JSON)
            Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/
            RFC8259, December 2017, <https://www.rfc-editor.org/info/
            rfc8259>.

[RFC8446]   Rescorla, E., "The Transport Layer Security (TLS)
            Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446,
            August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC9110]   Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke,
            Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/

RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[RFC9111]  Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022, <https://www.rfc-editor.org/info/rfc9111>.

[RFC9207]  Meyer zu Selhausen, K. and D. Fett, "OAuth 2.0 Authorization Server Issuer Identification", RFC 9207, DOI 10.17487/RFC9207, March 2022, <https://www.rfc-editor.org/info/rfc9207>.

[USASCII]  Institute, A. N. S., "Coded Character Set -- 7-bit American Standard Code for Information Interchange, ANSI X3.4", 1986.

[W3C.REC-xml-20081126] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and F. Yergeau, "Extensible Markup Language", November 2008, <https://www.w3.org/TR/REC-xml/REC-xml-20081126.xml>.

[WHATWG.URL] WHATWG, "URL", May 2022, <https://url.spec.whatwg.org/>.

## 12.2.  Informative References

[CSP-2]    "Content Security Policy Level 2", December 2016, <https://www.w3.org/TR/CSP2>.

[I-D.bradley-oauth-jwt-encoded-state] Bradley, J., Lodderstedt, T., and H. Zandbelt, "Encoding claims in the OAuth 2 state parameter using a JWT", Work in Progress, Internet-Draft, draft-bradley-oauth-jwt-encoded-state-09, 4 November 2018, <https://datatracker.ietf.org/doc/html/draft-bradley-oauth-jwt-encoded-state-09>.

[I-D.ietf-oauth-browser-based-apps] Parecki, A. and D. Waite, "OAuth 2.0 for Browser-Based Apps", Work in Progress, Internet-Draft, draft-ietf-oauth-browser-based-apps-13, 13 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-browser-based-apps-13>.

[I-D.ietf-oauth-dpop] Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)", Work in Progress, Internet-Draft, draft-ietf-oauth-dpop-14, 8 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop-14>.

**[I-D.ietf-oauth-rar]**
Lodderstedt, T., Richer, J., and B. Campbell,
"OAuth 2.0 Rich Authorization Requests", Work in
Progress, Internet-Draft, draft-ietf-oauth-rar-23, 30
January 2023, <https://datatracker.ietf.org/doc/html/
draft-ietf-oauth-rar-23>.

**[I-D.ietf-oauth-token-binding]** Jones, M., Campbell, B., Bradley, J.,
and W. Denniss, "OAuth 2.0 Token Binding", Work in
Progress, Internet-Draft, draft-ietf-oauth-token-
binding-08, 19 October 2018, <https://
datatracker.ietf.org/doc/html/draft-ietf-oauth-token-
binding-08>.

**[NIST800-63]** Burr, W., Dodson, D., Newton, E., Perlner, R., Polk,
T., Gupta, S., and E. Nabbus, "NIST Special Publication
800-63-1, INFORMATION SECURITY", December 2011, <http://
csrc.nist.gov/publications/>.

**[OMAP]**
Huff, J., Schlacht, D., Nadalin, A., Simmons, J.,
Rosenberg, P., Madsen, P., Ace, T., Rickelton-Abdi, C.,
and B. Boyer, "Online Multimedia Authorization Protocol:
An Industry Standard for Authorized Access to Internet
Multimedia Resources", August 2012, <https://
www.svta.org/product/online-multimedia-authorization-
protocol/>.

**[OpenID]**    Sakimura, N., Bradley, J., Jones, M., de Medeiros, B.,
and C. Mortimore, "OpenID Connect Core 1.0", November
2014, <https://openid.net/specs/openid-connect-
core-1_0.html>.

**[OpenID.Messages]** Sakimura, N., Bradley, J., Jones, M., de Medeiros,
B., Mortimore, C., and E. Jay, "OpenID Connect Messages
1.0", June 2012, <http://openid.net/specs/openid-connect-
messages-1_0.html>.

**[owasp_redir]** "OWASP Cheat Sheet Series - Unvalidated Redirects and
Forwards", 2020, <https://cheatsheetseries.owasp.org/
cheatsheets/
Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html>.

**[RFC6265]**   Barth, A., "HTTP State Management Mechanism", RFC 6265,
DOI 10.17487/RFC6265, April 2011, <https://www.rfc-
editor.org/info/rfc6265>.

**[RFC6819]**   Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth
2.0 Threat Model and Security Considerations", RFC 6819,

DOI 10.17487/RFC6819, January 2013, <https://www.rfc-editor.org/info/rfc6819>.

[RFC7009]   Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "OAuth 2.0 Token Revocation", RFC 7009, DOI 10.17487/RFC7009, August 2013, <https://www.rfc-editor.org/info/rfc7009>.

[RFC7519]   Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7591]   Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

[RFC7592]   Richer, J., Ed., Jones, M., Bradley, J., and M. Machulak, "OAuth 2.0 Dynamic Client Registration Management Protocol", RFC 7592, DOI 10.17487/RFC7592, July 2015, <https://www.rfc-editor.org/info/rfc7592>.

[RFC7636]   Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <https://www.rfc-editor.org/info/rfc7636>.

[RFC7662]   Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <https://www.rfc-editor.org/info/rfc7662>.

[RFC8414]   Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

[RFC8628]   Denniss, W., Bradley, J., Jones, M., and H. Tschofenig, "OAuth 2.0 Device Authorization Grant", RFC 8628, DOI 10.17487/RFC8628, August 2019, <https://www.rfc-editor.org/info/rfc8628>.

[RFC8705]   Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", RFC 8705, DOI 10.17487/RFC8705, February 2020, <https://www.rfc-editor.org/info/rfc8705>.

[RFC8707]   Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", RFC 8707, DOI 10.17487/

                    RFC8707, February 2020, <https://www.rfc-editor.org/info/
                    rfc8707>.

     [RFC9068]  Bertocci, V., "JSON Web Token (JWT) Profile for OAuth 2.0
                    Access Tokens", RFC 9068, DOI 10.17487/RFC9068, October
                    2021, <https://www.rfc-editor.org/info/rfc9068>.

     [RFC9126]  Lodderstedt, T., Campbell, B., Sakimura, N., Tonge, D.,
                    and F. Skokan, "OAuth 2.0 Pushed Authorization Requests",
                    RFC 9126, DOI 10.17487/RFC9126, September 2021, <https://
                    www.rfc-editor.org/info/rfc9126>.

## Appendix A.  Augmented Backus-Naur Form (ABNF) Syntax

   This section provides Augmented Backus-Naur Form (ABNF) syntax
   descriptions for the elements defined in this specification using
   the notation of [RFC5234]. The ABNF below is defined in terms of
   Unicode code points [W3C.REC-xml-20081126]; these characters are
   typically encoded in UTF-8. Elements are presented in the order
   first defined.

   Some of the definitions that follow use the "URI-reference"
   definition from [RFC3986].

   Some of the definitions that follow use these common definitions:

```
VSCHAR      = %x20-7E
NQCHAR      = %x21 / %x23-5B / %x5D-7E
NQSCHAR     = %x20-21 / %x23-5B / %x5D-7E
UNICODECHARNOCRLF = %x09 /%x20-7E / %x80-D7FF /
                   %xE000-FFFD / %x10000-10FFFF
```

   (The UNICODECHARNOCRLF definition is based upon the Char definition
   in Section 2.2 of [W3C.REC-xml-20081126], but omitting the Carriage
   Return and Linefeed characters.)

### A.1.  "client_id" Syntax

   The client_id element is defined in Section 2.4.1:

```
client-id    = *VSCHAR
```

### A.2.  "client_secret" Syntax

   The client_secret element is defined in Section 2.4.1:

```
client-secret = *VSCHAR
```

### A.3.  "response_type" Syntax

The response_type element is defined in Section 4.1.1 and Section 6.4:

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

### A.4.  "scope" Syntax

The scope element is defined in Section 3.2.2.1:

```
scope       = scope-token *( SP scope-token )
scope-token = 1*NQCHAR
```

### A.5.  "state" Syntax

The state element is defined in Section 4.1.1, Section 4.1.2, and Section 4.1.2.1:

```
state       = 1*VSCHAR
```

### A.6.  "redirect_uri" Syntax

The redirect_uri element is defined in Section 4.1.1, and Section 4.1.3:

```
redirect-uri      = URI-reference
```

### A.7.  "error" Syntax

The error element is defined in Sections Section 4.1.2.1, Section 3.2.3.1, 7.2, and 8.5:

```
error             = 1*NQSCHAR
```

### A.8.  "error_description" Syntax

The error_description element is defined in Sections Section 4.1.2.1, Section 3.2.3.1, and Section 5.3:

```
error-description = 1*NQSCHAR
```

### A.9.  "error_uri" Syntax

The error_uri element is defined in Sections Section 4.1.2.1, Section 3.2.3.1, and 7.2:

```
error-uri         = URI-reference
```

### A.10.  "grant_type" Syntax

The grant_type element is defined in Section [Section 3.2.2](#):

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

### A.11.  "code" Syntax

The code element is defined in [Section 4.1.3](#):

```
code        = 1*VSCHAR
```

### A.12.  "access_token" Syntax

The access_token element is defined in [Section 3.2.3](#):

```
access-token = 1*VSCHAR
```

### A.13.  "token_type" Syntax

The token_type element is defined in [Section 3.2.3](#), and [Section 6.1](#):

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

### A.14.  "expires_in" Syntax

The expires_in element is defined in [Section 3.2.3](#):

```
expires-in = 1*DIGIT
```

### A.15.  "refresh_token" Syntax

The refresh_token element is defined in [Section 3.2.3](#) and [Section 4.3](#):

```
refresh-token = 1*VSCHAR
```

### A.16.  Endpoint Parameter Syntax

The syntax for new endpoint parameters is defined in [Section 6.2](#):

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

### A.17.  "code_verifier" Syntax

ABNF for code_verifier is as follows.

```
code-verifier = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

### A.18.  "code_challenge" Syntax

ABNF for code_challenge is as follows.

```
code-challenge = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

## Appendix B.  Use of application/x-www-form-urlencoded Media Type

TBD: Describe OAuth's use of application/x-www-form-urlencoded encoding, both for URL parameters as well as for the client_id and secret encoding.

GitHub discussion: https://github.com/oauth-wg/oauth-v2-1/issues/128

## Appendix C.  Extensions

Below is a list of well-established extensions at the time of publication:

*[RFC8628]: OAuth 2.0 Device Authorization Grant

  -The Device Authorization Grant (formerly known as the Device Flow) is an extension that enables devices with no browser or limited input capability to obtain an access token. This is commonly used by smart TV apps, or devices like hardware video encoders that can stream video to a streaming video service.

*[RFC8414]: Authorization Server Metadata

  -Authorization Server Metadata (also known as OAuth Discovery) defines an endpoint clients can use to look up the information needed to interact with a particular OAuth server, such as the location of the authorization and token endpoints and the supported grant types.

*[RFC8707]: Resource Indicators

  -Provides a way for the client to explicitly signal to the authorization server where it intends to use the access token it is requesting.
```

*[RFC7591]: Dynamic Client Registration

  -Dynamic Client Registration provides a mechanism for
   programmatically registering clients with an authorization
   server.

*[RFC7592]: Dynamic Client Management

  -Dynamic Client Management provides a mechanism for updating
   dynamically registered client information.

*[RFC9068]: JSON Web Token (JWT) Profile for OAuth 2.0 Access
 Tokens

  -This specification defines a profile for issuing OAuth access
   tokens in JSON Web Token (JWT) format.

*[RFC8705]: Mutual TLS

  -Mutual TLS describes a mechanism of binding access tokens and
   refresh tokens to the clients they were issued to, as well as
   a client authentication mechanism, via TLS certificate
   authentication.

*[RFC7662]: Token Introspection

  -The Token Introspection extension defines a mechanism for
   resource servers to obtain information about access tokens.

*[RFC7009]: Token Revocation

  -The Token Revocation extension defines a mechanism for clients
   to indicate to the authorization server that an access token
   is no longer needed.

*[RFC9126]: Pushed Authorization Requests

  -The Pushed Authorization Requests extension describes a
   technique of initiating an OAuth flow from the back channel,
   providing better security and more flexibility for building
   complex authorization requests.

*[RFC9207]: Authorization Server Issuer Identification

  -The iss parameter in the authorization response indicates the
   identity of the authorization server to prevent mix-up attacks
   in the client.

*[I-D.ietf-oauth-rar]: Rich Authorization Requests

   -Rich Authorization Requests specifies a new parameter
    authorization_details that is used to carry fine-grained
    authorization data in the OAuth authorization request.

## Appendix D.  Acknowledgements

This specification is the work of the OAuth Working Group, and its
starting point was based on the contents of the following
specifications: OAuth 2.0 Authorization Framework (RFC 6749), OAuth
2.0 for Native Apps (RFC 8252), OAuth Security Best Current
Practice, and OAuth 2.0 for Browser-Based Apps. The editors would
like to thank everyone involved in the creation of those
specifications upon which this is built.

The editors would also like to thank the following individuals for
their ideas, feedback, corrections, and wording that helped shape
this version of the specification: Vittorio Bertocci, Michael Jones,
Justin Richer, Daniel Fett, Brian Campbell, Joseph Heenan, Roberto
Polli, Andrii Deinega, Falko, Michael Peck, and Bob Hamburg.

Discussions around this specification have also occurred at the
OAuth Security Workshop in 2021 and 2022. The authors thank the
organizers of the workshop (Guido Schmitz, Steinar Noem, and Daniel
Fett) for hosting an event that's conducive to collaboration and
community input.

## Appendix E.  Document History

[[ To be removed from the final specification ]]

-08

  *Updated acknowledgments

  *Swap "by a trusted party" with "by an outside party" in client ID
   definition

  *Replaced "verify the identity of the resource owner" with
   "authenticate"

  *Clarified refresh token rotation to match RFC6819

  *Added appendix to hold application/x-www-form-urlencoded examples

  *Fixed references to entries in appendix

  *Incorporated new "Phishing via AS" section from Security BCP

*Rephrase description of the motivation for client authentication

*Moved "scope" parameter in token request into specific grant
 types to match OAuth 2.0

*Updated Clickjacking and Open Redirection description from the
 latest version of the Security BCP

*Moved normative requirements out of authorization code security
 considerations section

*Security considerations clarifications, and removed a duplicate
 section

-07

*Removed "third party" from abstract

*Added MFA and passwordless as additional motiviations in
 introduction

*Mention PAR as one way redirect URI registration can happen

*Added a reference to requiring CORS headers on the token endpoint

*Updated reference to OMAP extension

*Fixed numbering in sequence diagram

-06

*Removed "credentialed client" term

*Simplified definition of "confidential" and "public" clients

*Incorporated the iss response parameter referencing RFC9207

*Added section on access token validation by the RS

*Removed requirement for authorization servers to support all 3
 redirect methods for native apps

*Fixes for some references

*Updates HTTP references to RFC 9110

*Clarifies "authorization grant" term

*Clarifies client credential grant usage

*Clean up authorization code diagram

*Updated reference for application/x-www-form-urlencoded and
   removed outdated note about it not being in the IANA registry

-05

  *Added a section about the removal of the implicit flow

  *Moved many normative requirements from security considerations
   into the appropriate inline sections

  *Reorganized and consolidated TLS language

  *Require TLS on redirect URIs except for localhost/custom URL
   scheme

  *Updated refresh token guidance to match security BCP

-04

  *Added explicit mention of not sending access tokens in URI query
   strings

  *Clarifications on definition of client types

  *Consolidated text around loopback vs localhost

  *Editorial clarifications throughout the document

-03

  *refactoring to collect all the grant types under the same top-
   level header in section 4

  *Better split normative and security consideration text into the
   appropriate places, both moving text that was really security
   considerations out of the main part of the document, as well as
   pulling normative requirements from the security considerations
   sections into the appropriate part of the main document

  *Incorporated many of the published errata on RFC6749

  *Updated references to various RFCs

  *Editorial clarifications throughout the document

-02

-01

-00

  *initial revision

**Authors' Addresses**

Dick Hardt
Hellō

Email: dick.hardt@gmail.com

Aaron Parecki
Okta

Email: aaron@parecki.com
URI: https://aaronparecki.com

Torsten Lodderstedt
yes.com

Email: torsten@lodderstedt.net