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

Abstract

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

Status of This Memo

Copyright Notice

Section 4.e of the Trust Legal Provisions and are provided without
warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

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

  *Third-party applications are required to store the resource owner's credentials for future use, typically a password in clear-text.

*Servers are required to support password authentication, despite
   the security weaknesses inherent in passwords.

  *Third-party applications gain overly broad access to the resource
   owner's protected resources, leaving resource owners without any
   ability to restrict duration or access to a limited subset of
   resources.

  *Resource owners cannot revoke access to an individual third party
   without revoking access to all third parties, and must do so by
   changing the third party's password.

  *Compromise of any third-party application results in compromise
   of the end-user's password and all of the data protected by that
   password.

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

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

For example, an end-user (resource owner) can grant a printing
service (client) access to her protected photos stored at a photo-
sharing service (resource server), without sharing her username and
password with the printing service. Instead, she authenticates
directly with a server trusted by the photo-sharing service
(authorization server), which issues the printing service
delegation- specific credentials (access token).

This specification is designed for use with HTTP ([RFC7230]). 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. 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 extension
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 ---|              |
+--------+                            +--------------+
```
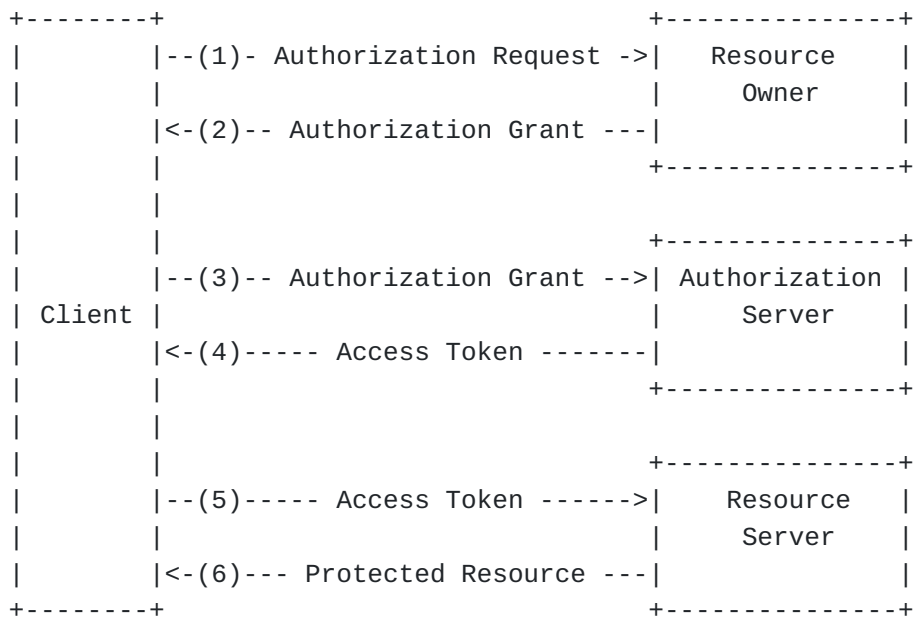
Figure 1: Abstract Protocol Flow

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

### 1.3.1.  Authorization Code

The authorization code is obtained by using an authorization server
as an intermediary between the client and resource owner. Instead of
requesting authorization directly from the resource owner, the
client directs the resource owner to an authorization server (via
its user-agent as defined in [[RFC7231](#)]), which in turn directs the
resource owner back to the client with the authorization code.

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

The authorization code provides a few important security benefits,
such as the ability to authenticate the client, 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.  Client Credentials

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

## 1.4.  Access Token

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 opaque to the client, but depending on the
authorization server, may be parseable by the resource server.

Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server.

The token may denote an identifier used to retrieve the authorization information or may self-contain the authorization information in a verifiable manner (i.e., a token string consisting of some data and a signature). One example of a structured token format is [I-D.ietf-oauth-access-token-jwt], 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 a token. This is typically referred to as a sender-constrained access token, such as 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.

## 1.5.  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. If the authorization server issues a refresh token, it is included when issuing an access token (i.e., step (4) in Figure 2).

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

```
  +--------+                                        +---------------+
  |        |--(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).

1.6. TLS Version

Whenever Transport Layer Security (TLS) is used by this
specification, the appropriate version (or versions) of TLS will
vary over time, based on the widespread deployment and known
security vulnerabilities. At the time of this writing, TLS version
1.3 [RFC8446] is the most recent version.

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

1.7. 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 9.7.2 for
details.

1.8. 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.

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

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".

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

2. Client Registration

Before initiating the protocol, the client registers with the
authorization server. The means through which the client registers
with the authorization server are beyond the scope of this
specification but typically involve end-user interaction with an
HTML registration form, 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 its client redirect URIs as described in Section 3.1.2,
   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

Clients are identified at the authorization server by a client_id.
It is, for example, used by the authorization server to determine
the set of redirect URIs this client can use.

Clients requiring a higher level of confidence in their identity by the authorization server use credentials to authenticate with the authorization server. Such credentials are either issued by the authorization server or registered by the developer of the client with the authorization server.

OAuth 2.1 defines three client types:

"confidential": Clients that have credentials and their identity has been confirmed by the AS are designated as "confidential clients"

"credentialed": Clients that have credentials and their identity has been not been confirmed by the AS are designated as "credentialed clients"

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

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

Authorization servers SHOULD consider the level of confidence in a client's identity when deciding whether they allow such a client access to more critical functions, such as the Client Credentials grant type.

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

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

"native application": A native application is a public client installed and executed on the device used by the resource owner.

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

2.2. Client Identifier

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

Authorization servers SHOULD NOT allow clients to choose or influence their client_id value. See Section 9.6 for details.

2.3. Client Authentication

If the client has credentials, (is a confidential or credentialed client), the client and authorization server establish a client authentication method suitable for the security requirements of the authorization server. The authorization server MAY accept any form of client authentication meeting its security requirements.

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

Authorization servers SHOULD use client authentication if possible.

It is RECOMMENDED to use asymmetric (public-key based) methods for client authentication such as mTLS [RFC8705] or "private_key_jwt" [OpenID]. 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.

The authorization server MAY establish a client authentication method with public clients, which converts them to credentialed clients. However, the authorization server MUST NOT rely on

credentialed client authentication for the purpose of identifying the client.

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

2.3.1. Client Password

Clients in possession of a client password, also known as a client secret, MAY use the HTTP Basic authentication scheme as defined in [RFC2617] 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
```

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

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

"client_secret": REQUIRED. The client secret.

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

For example, a request to refresh an access token (Section 6) using the body 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
```

The authorization server MUST require the use of TLS as described in [Section 1.6](#) when sending requests using password authentication.

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

2.3.2. Other Authentication Methods

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 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.

2.4. Unregistered Clients

This specification does not exclude the use of unregistered clients. However, the use of such 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 verify the identity of the resource owner. The way in which the authorization server authenticates the resource owner (e.g., username and password login, session cookies) is beyond the scope of this specification.

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

The endpoint URI MAY include an "application/x-www-form-urlencoded" formatted (per Appendix B) query component ([RFC3986] Section 3.4), which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

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

The authorization server MUST support the use of the HTTP GET method [RFC7231] for the authorization endpoint and MAY support the use of the POST method as well.

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

3.1.1. Response Type

The authorization endpoint is used by the authorization code flow. The client informs the authorization server of the desired response type using the following parameter:

"response_type": REQUIRED. The value MUST be code for requesting an authorization code as described by Section 4.1.1, or a registered extension value as described by Section 8.4.

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

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

3.1.2. Redirection Endpoint

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

The authorization server MUST compare the two URIs using simple string comparison as defined in [RFC3986], Section 6.2.1.

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 (per Appendix B) query component ([RFC3986] Section 3.4), which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

3.1.2.1. Endpoint Request Confidentiality

The redirection endpoint SHOULD require the use of TLS as described in Section 1.6 when the requested response type is code, or when the redirection request will result in the transmission of sensitive credentials over an open network. If TLS is not available, the authorization server SHOULD warn the resource owner about the insecure endpoint prior to redirection (e.g., display a message during the authorization request).

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

3.1.2.2. Registration Requirements

The authorization server MUST require all clients to register one or more complete redirect URIs prior to utilizing the authorization endpoint. The client MAY use the state request parameter to achieve per-request customization if needed.

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

Lack of requiring registration of redirect URIs enables an attacker to use the authorization endpoint as an open redirector as described in [Section 9.18](#).

3.1.2.3. Dynamic Configuration

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

3.1.2.4. 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.

3.1.2.5. 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.

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.

3.2. Token Endpoint

The token endpoint is used by the client to obtain an access token by presenting its authorization grant or refresh token.

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, or in the authorization server's metadata document ([RFC8414]).

The endpoint URI MAY include an application/x-www-form-urlencoded formatted (per Appendix B) query component ([RFC3986] Section 3.4), which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

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

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

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

3.2.1. Client Authentication

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

  *Enforcing the binding of refresh tokens and authorization codes to the client they were issued to. Client authentication is critical when an authorization code is transmitted to the redirection endpoint over an insecure channel.

  *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.

3.3. 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 to inform the client of the actual scope granted.

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

4. Obtaining Authorization

To request an access token, the client obtains authorization from the resource owner. The authorization is expressed in the form of an authorization grant, which the client uses to request the access token. OAuth defines two grant types: authorization code and client credentials. 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.

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

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

Note: The lines illustrating steps (1), (2), and (3) are broken into
two parts as they pass through the user-agent.

                    Figure 3: Authorization Code Flow

   The flow illustrated in Figure 3 includes the following steps:

   (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.

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

Clients MUST use code_challenge and code_verifier and authorization
servers MUST enforce their use except under the conditions described
in Section 9.8. In this case, using and enforcing code_challenge and
code_verifier as described in the following is still RECOMMENDED.

4.1.1.1. Client Creates a Code Verifier

The client first creates a code verifier, code_verifier, for each
Authorization Request, in the following manner:

code_verifier = high-entropy cryptographic random STRING using the
unreserved characters `[A-Z] / [a-z] / [0-9] / "-" / "." / "_" / "~"`
from Section 2.3 of {{RFC3986}}, with a minimum length of 43 characters
and a maximum length of 128 characters.

ABNF for code_verifier is as follows.

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

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.

4.1.1.2. Client Creates the Code Challenge

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

plain
  code_challenge = code_verifier

S256
  code_challenge = BASE64URL-ENCODE(SHA256(ASCII(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 and know via out-of-band configuration or via Authorization Server Metadata ([RFC8414]) that the server supports plain.

The plain transformation is for compatibility with existing deployments and for constrained environments that can't use the S256 transformation.

ABNF for code_challenge is as follows.

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

4.1.1.3. Client Initiates the Authorization Request

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

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

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

"code_challenge": REQUIRED or RECOMMENDED (see Section 9.8). 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 3.1.2.

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

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

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

   For example, the client directs the user-agent to make the following
   HTTP request using TLS (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. 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 generated by the
   authorization server. The authorization code MUST expire shortly
   after it is issued to mitigate the risk of leaks. A maximum
   authorization code lifetime of 10 minutes is RECOMMENDED. The
   client MUST NOT use the authorization code more than once. If an
   authorization code is used more than once, the authorization
   server MUST deny the request and SHOULD revoke (when possible)
   all tokens previously issued based on that authorization code.
   The authorization code is bound to the client identifier and
   redirect URI.

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

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

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

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

When the server issues the authorization code in the authorization
response, it MUST associate the code_challenge and
code_challenge_method values with the authorization code so it can
be verified later.

The code_challenge and code_challenge_method values may be stored in
encrypted form in the code itself, but could alternatively be stored
on the server associated with the code. The server MUST NOT include
the code_challenge value in client requests in a form that other
entities can extract.

The exact method that the server uses to associate the
code_challenge with the issued code is out of scope for this
specification.

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 9.8 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.

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

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

4.1.3. Access Token Request

The client makes a request to the token endpoint by 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 entity-body:

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

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

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

"client_id":
                  REQUIRED, if the client is not authenticating with the
      authorization server as described in [Section 3.2.1](#).

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

   Confidential or credentialed clients MUST authenticate with the
   authorization server as described in [Section 3.2.1](#).

   For example, the client makes the following HTTP request using TLS
   (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 and credentialed
      clients (or clients with other authentication requirements),

     *authenticate the client if client authentication is included,

     *ensure that the authorization code was issued to the
      authenticated confidential or credentialed 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 4.1.1.3, and if included ensure
that their values are identical.

4.1.4. Access Token Response

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

An example successful response:

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

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

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
or credentialed clients.

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

                  Figure 4: Client Credentials Flow

The flow 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. Authorization Request and Response

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

4.2.2. Access Token Request

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

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

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

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

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

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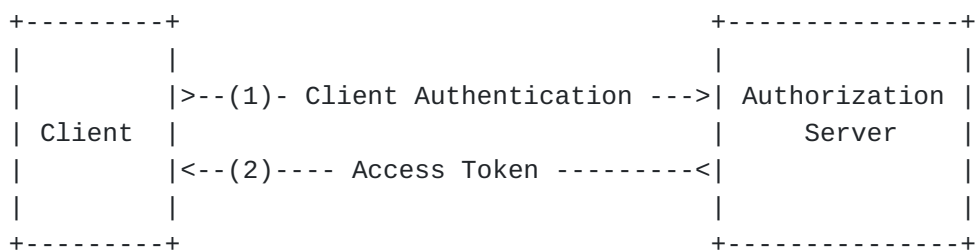POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
```

The authorization server MUST authenticate the client.

4.2.3. Access Token Response

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

An example successful response:

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

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

4.3. 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 using TLS (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 5.1. If the request failed client
authentication or is invalid, the authorization server returns an
error response as described in Section 5.2.

5. Issuing an Access Token

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

5.1. Successful Response

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

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

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

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

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

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

The parameters are included in the entity-body of the HTTP response using the application/json media type as defined by [RFC7159]. 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 [RFC7234] with a value of no-store in any response containing tokens, credentials, or other sensitive information, as well as the Pragma response header field [RFC7234] with a value of no-cache.

For example:

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

{
  "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.

5.2. 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 entity-body 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
Pragma: no-cache

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

6. Refreshing an Access Token

   Authorization servers SHOULD determine, based on a risk assessment,
   whether to issue refresh tokens to a certain client. If the
   authorization server decides not to issue refresh tokens, the client
   MAY refresh access tokens by utilizing other grant types, such as
   the authorization code grant type. 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.

   If the authorization server issued a refresh token to the client,
   the client makes a refresh request to the token endpoint by adding
   the following parameters using the application/x-www-form-urlencoded
   format per Appendix B with a character encoding of UTF-8 in the HTTP
   request entity-body:

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

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

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

   Because refresh tokens are typically long-lasting credentials used
   to request additional access tokens, the refresh token is bound to
   the client to which it was issued. Confidential or credentialed
   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
```

The authorization server MUST:

   *require client authentication for confidential or credentialed
    clients

   *authenticate the client if client authentication is included and
    ensure that the refresh token was issued to the authenticated
    client, and

   *validate the refresh token.

6.1. Refresh Token Protection

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. 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.

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

The authorization server MAY issue a new refresh token, in which
case the client MUST discard the old refresh token and replace it
with the new refresh token. The authorization server MAY revoke the

old refresh token after issuing a new refresh token to the client.
If a new refresh token is issued, the refresh token scope MUST be
identical to that of the refresh token included by the client in the
request.

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 fresh
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).

7. 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, such as using Token Introspection [RFC7662] or
a structured access token format such as a JWT [I-D.ietf-oauth-
access-token-jwt].

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

7.1. 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.

7.2. 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 extended to include proof-of-possession
techniques by other specifications.

7.2.1. Authenticated Requests

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

7.2.1.1. Authorization Request Header Field

When sending the access token in the Authorization request header
field defined by HTTP/1.1 [RFC2617], 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.

7.2.1.2. Form-Encoded Body Parameter

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

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

  *The entity-body follows the encoding requirements of the application/x-www-form-urlencoded content-type as defined by HTML 4.01 [W3C.REC-html401-19991224].

  *The HTTP request entity-body is single-part.

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

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

The entity-body 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.

7.2.2. 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 [RFC2617].

All challenges defined by this specification 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 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 [RFC2617]. The realm
attribute MUST NOT appear more than once.

The scope attribute is defined in Section 3.3. 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
7.2.3. 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 Appendixes A.7 and 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"
```

7.2.3. 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 than
   provided by the access token. The resource server SHOULD respond
   with the HTTP 403 (Forbidden) status code and MAY include the
   scope attribute with the scope necessary to access the protected
   resource.

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"
```

7.3. 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 7.2.3.

7.3.1. Extension Token Types

[RFC6750] establishes a common registry in Section 11.4 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.

7.4. Access Token Security Considerations

7.4.1. 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].

7.4.1.1. Token manufacture/modification

An attacker may generate a bogus 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.

7.4.1.2. Token disclosure

Tokens may contain authentication and attribute statements that
include sensitive information.

7.4.1.3. Token redirect

An attacker uses a 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.

7.4.1.4. Token replay

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

7.4.2. Threat Mitigation

A large range of threats can be mitigated by protecting the contents
of the token by using a digital signature. Alternatively, a bearer
token can contain a reference to authorization information, rather
than encoding the information directly. Such references MUST be
infeasible for an attacker to guess; using a reference may require
an extra interaction between a server and the token issuer to
resolve the reference to the authorization information. The
mechanics of such an interaction are not defined by this
specification.

This document does not specify the encoding or the contents of the
token; hence, detailed recommendations about the means of
guaranteeing token integrity protection are outside the scope of
this document. The token integrity protection MUST be sufficient to
prevent the token from being modified.

To deal with token redirect, 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.

The authorization server MUST implement TLS. Which version(s) ought to be implemented will vary over time and will depend on the widespread deployment and known security vulnerabilities at the time of implementation.

To protect against token disclosure, confidentiality protection MUST be applied using TLS with a ciphersuite that provides confidentiality and integrity protection. This requires that the communication interaction between the client and the authorization server, as well as the interaction between the client and the resource server, utilize confidentiality and integrity protection. Since TLS is mandatory to implement and to use with this specification, it is the preferred approach for preventing token disclosure via the communication channel. For those cases where the client is prevented from observing the contents of the token, token encryption MUST be applied in addition to the usage of TLS protection. As a further defense against token disclosure, the client MUST validate the TLS certificate chain when making requests to protected resources, including checking the Certificate Revocation List (CRL) [RFC5280].

Cookies are typically transmitted in the clear. Thus, 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 token between the front-end and back-end servers; encryption of the token is one such possible measure.

To deal with token capture and replay, the following recommendations are made: First, the lifetime of the token MUST be limited; one means of achieving this is by putting a validity time field inside the protected part of the token. Note that using short-lived (one hour or less) tokens reduces the impact of them being leaked. Second, confidentiality protection of the exchanges between the client and the authorization server and between the client and the resource server MUST be applied. As a consequence, no eavesdropper along the communication path is able to observe the token exchange. Consequently, such an on-path adversary cannot replay the token. Furthermore, when presenting the token to a resource server, the client MUST verify the identity of that resource server, as per

Section 3.1 of "HTTP Over TLS" [RFC2818]. Note that the client MUST validate the TLS certificate chain when making these requests to protected resources. Presenting the token to an unauthenticated and unauthorized resource server or failing to validate the certificate chain will allow adversaries to steal the token and gain unauthorized access to protected resources.

7.4.3. Summary of Recommendations

7.4.3.1. 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.

7.4.3.2. 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.

7.4.3.3. 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.

7.4.3.4. 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.

7.4.3.5. Issue short-lived bearer tokens

Token servers SHOULD issue short-lived (one hour or less) 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.

7.4.3.6. Issue scoped bearer tokens

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

7.4.3.7. 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.

7.4.4. Token Replay Prevention

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 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.

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.

7.4.5. 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.

8. Extensibility

8.1. 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
```

8.2. 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_').

8.3. 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].

8.4. 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.

8.5. 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 5.2),
or the resource access error response (Section 7.3), 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
```

9. 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].

9.1. Client Authentication

Authorization servers SHOULD use client authentication if possible.

It is RECOMMENDED to use asymmetric (public-key based) methods for
client authentication such as mTLS [RFC8705] or private_key_jwt
[OpenID]. 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.

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

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

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. For example, authentication of a dynamically registered

client just ensures the authorization server it is talking to the
same client again. In contrast, if there is a web 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 access
more sensible services or to use the client credential grant type.

9.1.1. 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 9.3.1).

9.2. 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.

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

For private-use URI scheme-based redirects, authorization servers
SHOULD enforce the requirement in Section 10.3.1 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, requiring a URI
scheme based on a domain name that is under the control of the app
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.

Authorization servers MAY request the inclusion of other platform-specific information, such as the app package or bundle name, or other information that may be useful for verifying the calling app's identity on operating systems that support such functions.

9.3. Client Impersonation

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

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

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

9.3.1. 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.

9.4. Access Tokens

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

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

9.4.1. 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 less rights 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.

9.4.2. Access Token Replay Prevention

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.

Authorization and resource servers SHOULD use mechanisms for sender-
constrained access tokens to prevent token replay as described in
(#pop_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 token at the recipient (e.g., a resource
server). The use of Mutual TLS for OAuth 2.0 [RFC8705] is
RECOMMENDED.

9.5. Refresh Tokens

Authorization servers MAY issue refresh tokens to clients.

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

The authorization server MUST verify the binding between the refresh
token and client identity whenever the client identity can be
authenticated. When client authentication is not possible, the
authorization server SHOULD issue sender-constrained refresh tokens
or use refresh token rotation as described in
(#refresh_token_protection).

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

9.6. Client Impersonating Resource Owner

Resource servers may make access control decisions based on the
identity of the resource owner as communicated in the sub claim
returned by the authorization server in a token introspection
response [RFC7662] or other mechanisms. If a client is able to
choose its own client_id during registration with the authorization
server, then there is a risk that it can register with the same sub
value as a privileged user. A subsequent access token obtained under
the client credentials grant may be mistaken for an access token

authorized by the privileged user if the resource server does not perform additional checks.

Authorization servers SHOULD NOT allow clients to influence their client_id or sub value 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.

9.7. Protecting Redirect-Based Flows

When comparing client redirect URIs against pre-registered URIs, authorization servers MUST utilize exact string matching. This measure contributes to the prevention of leakage of authorization codes and access tokens (see (#insufficient_uri_validation)). It can also help to detect mix-up attacks (see (#mix_up)).

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

Clients MUST prevent Cross-Site Request Forgery (CSRF). 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 the CSRF protection provided by that mechanism. In OpenID Connect flows, 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 (#csrf_countermeasures)).

In order to prevent mix-up attacks (see (#mix_up)), 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 request was received from the correct authorization server. Clients MUST ensure that the subsequent 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.

An AS that redirects a request potentially containing user credentials MUST avoid forwarding these user credentials accidentally (see Section 9.7.2 for details).

9.7.1. Loopback Redirect Considerations in Native Apps

Loopback interface redirect URIs use the http scheme (i.e., without Transport Layer Security (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.

While redirect URIs using localhost (i.e., http://localhost:{port}/{path}) function similarly to loopback IP redirects described in Section 10.3.3, 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.

9.7.2. HTTP 307 Redirect

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

At the authorization endpoint, a typical protocol flow is that the AS prompts the user to enter her credentials in a form that is then submitted (using the HTTP 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 HTTP POST 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 [RFC7231], Section 6.4.7. 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 body.

In the HTTP standard [RFC7231], 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.

9.8. Authorization Codes

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

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

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

Clients MUST prevent injection (replay) of authorization codes into the authorization response by attackers. To this end, 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 or credentialed client.

   *In the specific deployment and the specific request, there is
    reasonable assurance for 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] 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.

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
(#secmodel)) can break the security provided by this mechanism.
Currently, S256 is the only such method.

When an authorization code arrives at the token endpoint, the
authorization server MUST do the following check:

1. If there was a code_challenge in the authorization request for
   which this code was issued, there must be a code_verifier in
   the token request, and it MUST be verified according to the
   steps in Section 4.1.3. (This is no change from the current
   behavior in [RFC7636].)

2. If there was no code_challenge in the authorization request,
   any request to the token endpoint containing a code_verifier
   MUST be rejected.

Authorization servers MUST support the code_challenge and
code_verifier parameters.

Authorization servers MUST provide a way to detect their support for
the code_challenge mechanism. To this end, they MUST either (a)
publish the element code_challenge_methods_supported in their AS
metadata ([RFC8414]) containing the supported code_challenge_methods
(which can be used by the client to detect support) or (b) provide a
deployment-specific way to ensure or determine support by the AS.

9.9. Request Confidentiality

Access tokens, refresh tokens, authorization codes, and client
credentials MUST NOT be transmitted in the clear.

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

9.10. Ensuring Endpoint Authenticity

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

TLS certificate as defined by [RFC6125] and in accordance with its
requirements for server identity authentication.

9.11. 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.

9.12. 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.

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

9.13. 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.

9.14. 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.

9.15. 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.

9.16. Clickjacking

As described in Section 4.4.1.9 of [RFC6819], the authorization
request is susceptible to clickjacking. 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.

9.17. 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.

9.18. Open Redirectors

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.

9.18.1. 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 and access tokens, as described in (#redir_uri_open_redir). 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].

9.18.2. 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 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.

The AS MUST take precautions to prevent this threat. Based on its risk assessment, the AS needs to decide whether it can trust the redirect URI and 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.

9.19. 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 9.2, specifically that authorization servers reject requests with URIs that don't match what was registered, is also required to prevent such attacks.

9.20. 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 credential, 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.

9.21. Other Recommendations

Authorization servers SHOULD NOT allow clients to influence their client_id or sub value or any other claim if that can cause confusion with a genuine resource owner (see (#client_impersonating)).

10. 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 9.20 for a deeper analysis of the drawbacks of using embedded user-agents for OAuth.

Native app authorization requests that use the browser are more secure and can take advantage of the user's authentication state. 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.

10.1. 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 10.2) and using a redirect URI that will return the authorization response back to the native app (defined in Section 10.3).

10.2. 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 10.3. 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.

10.3. 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.

To fully support native apps, authorization servers MUST offer at
least the three redirect URI options described in the following
subsections to native apps. Native apps MAY use whichever redirect
option suits their needs best, taking into account platform-specific
implementation details.

10.3.1. 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.

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.

10.3.2. Claimed "https" Scheme URI Redirection

Some operating systems allow apps to claim https scheme [RFC7230]
URIs 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 9.2 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.

10.3.3. 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

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.

11. 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.

12. 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 should either be sender-constrained or one-time
   use as per Section 4.12.2 of [I-D.ietf-oauth-security-topics]

13. 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.

14. References

14.1. Normative References

[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-15, 5 April 2020, <http://www.ietf.org/internet-
          drafts/draft-ietf-oauth-security-topics-15.txt>.

[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>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/
          RFC2818, May 2000, <https://www.rfc-editor.org/info/
          rfc2818>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO
          10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November
          2003, <https://www.rfc-editor.org/info/rfc3629>.

[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>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S.,
           Housley, R., and W. Polk, "Internet X.509 Public Key
           Infrastructure Certificate and Certificate Revocation
           List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May
           2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and
           Verification of Domain-Based Application Service Identity
           within Internet Public Key Infrastructure Using X.509
           (PKIX) Certificates in the Context of Transport Layer
           Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March
           2011, <https://www.rfc-editor.org/info/rfc6125>.

[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>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext
           Transfer Protocol (HTTP/1.1): Semantics and Content", RFC
           7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke,
           Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching",
           RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://www.rfc-editor.org/info/rfc7234>.

[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>.

[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>.

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

[W3C.REC-html401-19991224] Raggett, D., Hors, A., and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, 24 December 1999, <http://www.w3.org/TR/1999/REC-html401-19991224>.

[W3C.REC-xml-20081126] Bray, T., Paoli, J., Sperberg-McQueen, M., Maler, E., and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", World Wide Web Consortium Recommendation REC-xml-20081126, 26 November 2008, <http://www.w3.org/TR/2008/REC-xml-20081126>.

14.2. 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, <http://www.ietf.org/internet-drafts/draft-bradley-oauth-jwt-encoded-state-09.txt>.

[I-D.ietf-oauth-access-token-jwt]
            Bertocci, V., "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens", Work in Progress, Internet-Draft, draft-ietf-oauth-access-token-jwt-07, 27 April 2020, <http://www.ietf.org/internet-drafts/draft-ietf-oauth-access-token-jwt-07.txt>.

[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-06, 5 April 2020, <http://
            www.ietf.org/internet-drafts/draft-ietf-oauth-browser-
            based-apps-06.txt>.

[I-D.ietf-oauth-dpop] Fett, D., Campbell, B., Bradley, J.,
            Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0
            Demonstration of Proof-of-Possession at the Application
            Layer (DPoP)", Work in Progress, Internet-Draft, draft-
            ietf-oauth-dpop-01, 1 May 2020, <http://www.ietf.org/
            internet-drafts/draft-ietf-oauth-dpop-01.txt>.

[I-D.ietf-oauth-par]
            Lodderstedt, T., Campbell, B., Sakimura, N., Tonge, D.,
            and F. Skokan, "OAuth 2.0 Pushed Authorization Requests",
            Work in Progress, Internet-Draft, draft-ietf-oauth-
            par-02, 10 July 2020, <http://www.ietf.org/internet-
            drafts/draft-ietf-oauth-par-02.txt>.

[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-01, 19 February 2020,
            <http://www.ietf.org/internet-drafts/draft-ietf-oauth-
            rar-01.txt>.

[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, <http://www.ietf.org/internet-drafts/draft-ietf-
            oauth-token-binding-08.txt>.

[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", April 2012, <https://www.oatc.us/
            Standards/Download-Standards>.

[OpenID]   Sakimora, 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>.

[RFC7230]   Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[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>.

[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>.

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 2.3.1:

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

A.2. "client_secret" Syntax

The client_secret element is defined in Section 2.3.1:

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

A.3. "response_type" Syntax

The response_type element is defined in Section 3.1.1 and Section 8.4:

```
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 3.3:

```
 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 5.2, 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 5.2](#), and [Section 7.3](#):

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

A.9. "error_uri" Syntax

The error_uri element is defined in Sections [Section 4.1.2.1](#), [Section 5.2](#), and 7.2:

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

A.10. "grant_type" Syntax

The grant_type element is defined in Sections [Section 4.1.3](#), [Section 4.2.3](#), [Section 4.2.2](#), [Section 4.3](#), and [Section 6](#):

```
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 4.2.3](#) and [Section 5.1](#):

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

A.13. "token_type" Syntax

The token_type element is defined in [Section 5.1](#), and [Section 8.1](#):

```
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 5.1](#):

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

A.15. "refresh_token" Syntax

The refresh_token element is defined in [Section 5.1](#) and [Section 6](#):

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

A.16. Endpoint Parameter Syntax

The syntax for new endpoint parameters is defined in [Section 8.2](#):

```
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

At the time of publication of this specification, the application/x-www-form-urlencoded media type was defined in Section 17.13.4 of [W3C.REC-html401-19991224] but not registered in the IANA MIME Media Types registry (http://www.iana.org/assignments/media-types). Furthermore, that definition is incomplete, as it does not consider non-US-ASCII characters.

To address this shortcoming when generating payloads using this media type, names and values MUST be encoded using the UTF-8 character encoding scheme [RFC3629] first; the resulting octet sequence then needs to be further encoded using the escaping rules defined in [W3C.REC-html401-19991224].

When parsing data from a payload using this media type, the names and values resulting from reversing the name/value encoding consequently need to be treated as octet sequences, to be decoded using the UTF-8 character encoding scheme.

For example, the value consisting of the six Unicode code points (1)
U+0020 (SPACE), (2) U+0025 (PERCENT SIGN), (3) U+0026 (AMPERSAND),
(4) U+002B (PLUS SIGN), (5) U+00A3 (POUND SIGN), and (6) U+20AC
(EURO SIGN) would be encoded into the octet sequence below (using
hexadecimal notation):

20 25 26 2B C2 A3 E2 82 AC

and then represented in the payload as:

+%25%26%2B%C2%A3%E2%82%AC

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.

*[I-D.ietf-oauth-access-token-jwt]: 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.

*[I-D.ietf-oauth-par]: Pushed Authorization Requests

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

*[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

TBD

Authors' Addresses

Dick Hardt
SignIn.Org

Email: dick.hardt@gmail.com

Aaron Parecki
Okta

Email: aaron@parecki.com

URI: https://aaronparecki.com

Torsten Lodderstedt

yes.com

Email: torsten@lodderstedt.net