WLCG Common JWT Profiles

Authored by the WLCG AuthZ Working Group

Version History:

Date	Version	Comment
17.09.2019	0.1	Final version presented to MB
25.09.2019	1.0	Version published on Zenodo

Introduction

This document describes how WLCG users may use the available geographically distributed resources without X.509 credentials. In this model, clients are issued with bearer tokens; these tokens are subsequently used to interact with resources. The tokens may contain authorization groups and/or capabilities, according to the preference of the VO, applications and relying parties.

Wherever possible, this document builds on existing standards when describing profiles to support current and anticipated WLCG usage. In particular, three major technologies are identified as providing the basis for this system: OAuth2 (RFC 6749 & RFC 6750), OpenID Connect and JSON Web Tokens (RFC 7519). Additionally, trust roots are established via OpenID Discovery or OAuth2 Authorization Server Metadata (RFC 8414). This document provides a profile for OAuth2 Access Tokens and OIDC ID Tokens. **The WLCG Token Profile version described by this document is "1.0".**

The profile for the usage of JSON Web Tokens (RFC 7519) supports distributed authentication and authorization within the WLCG. The JWT profile is meant as a mechanism to transition away from the existing GSI-based (Globus) system where authentication is based on X509 proxy certificates and authorization is based on VOMS extensions and identity mapping.

The trust model used in this profile is VO-centric and uses the concept of transitive trust: an individual establishes an identity within the VO (through an identity proofing mechanism not described here) and any authentication mechanism happens within the scope of the VO. This is in strong contrast to the current X509-based system where a global identity is established completely orthogonal to the VO.

The WLCG has identified two strong use cases for these profiles: issuing information about an identity and issuing bearer-token-based authorizations. Identities are typically needed within a VO's services, which might provide different views or authorization based on the individual's identity within the VO.

We do not see the VO-based identity being authenticated from a bespoke username/password for the WLCG, but rather through the various global identity federations in use by the community. For CERN-centric VOs, this may be as simple as integrating with CERN SSO; however, it is considered out-of-scope for this document.

Note that the authorization model is VO-centric: the VO is authorizing access to its distributed pool of resources. The user authentication and the resource authorization are independent in such a way that a user authenticating with e.g. a SAML (Security Assertion Markup Language) assertion issued by their home organisation with a certain validity period may be provisioned by the VO with an OAuth2 Access Token with a different validity period.

Although VOs could implement their own solutions according to an agreed specification, it is hoped that a common implementation can be used (analogous to how VOMS-Admin is operated at CERN).

One item not addressed in detail in this document is how the issuer decides on whether to issue tokens - and how the token transits from the issuer to the client. It is envisioned that Access and ID Tokens linked to a user identity be acquired through OIDC flows and that WLCG VOs will decide on their issuing policy. An exception to this is the OAuth2 Client Credential Authentication flow, since there is no user identity associated with the client. OAuth2 flows may be used following OIDC authentication, for example for token exchange or introspection. A description of these flows is provided in the Appendix.

Glossary

Term	**Definition**	**WLCG Example (if applicable)**
OAuth 2.0	OAuth 2.0 is the industry-standard protocol for authorization. OAuth 2.0 supersedes the work done on the original OAuth protocol created in 2006. OAuth 2.0 is used for delegating authorization to a client (defined below). In many implementations, the OAuth 2.0-issued Access Tokens build upon JSON Web Tokens (RFC 7519).	
OIDC (or OpenID Connect)	OpenID Connect (OIDC) is an authentication layer on top of OAuth 2.0. It leverages a specific OAuth 2.0 flow with the aim of providing authentication information and relevant identity attributes. OIDC flows may issue OAuth2 Access Tokens, Refresh Tokens as well as ID Tokens.	
Token	JSON Web Token (JWT). A string representing a set of claims (pieces of information about a subject) as a JSON object that is encoded in a JWS or JWE, enabling the claims to be digitally signed or MACed and/or encrypted. JSON Web Token - RFC 7519	An OIDC or OAuth Token issued by the VO
Access Token	Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client[1].	

[1] https://tools.ietf.org/html/rfc6749#section-1.4

ID Token	A JWT specified by OIDC that contains user information, represented in the form of claims[2].	
Issuer	Any token issuer, this refers to both Authorization Servers and OpenID Providers.	VO https url
Authorization Server	The entity which produces (“issues”) the token. For WLCG authorization, this is a service run by the VO that is asserting the identity or the authorization to access the VO’s resources. This term is defined by OAuth2 and may be referred to as the Authorization Server.	Future WLCG VO Identity and Attribute Management Service
OpenID Connect Provider (OP)	A specific implementation of the OAuth Authorization server, which provides user authentication and represents an entity that offers user authentication as a service. It provides additional functionality, such as a /userinfo endpoint. This term is defined by OpenID Connect.	
Client	An application making protected resource requests on behalf of the user 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).	E.g. HTCondor submit host or an experiment framework
Relying Party (RP)	Can be applied to both OAuth client and resource provider roles; it is an application that outsources its user authentication function to an external Identity Provider. This term has been adopted by OpenID Connect. It is often used synonymously with “Client”.	E.g. PanDA framework
Bearer	A user’s agent that holds the token and is able to send it securely to a third party.	E.g. a job
Resource Provider	An entity that receives a Token, validates it, and decides whether to provide the bearer access to a	E.g. a Storage Element

[2] Note: in the OpenID Connect core specification, the ID token is intended primarily to contain information about the authentication, while profile information such as the user’s name and email is typically retrieved via the UserInfo endpoint. Since this puts a strain on the OP, we prefer to make - as much as possible - use of self-contained tokens, and return all the claims in the ID token

	corresponding resource. This term is defined by OAuth2 and may be referred to as the Resource Server.	
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 term is defined by OAuth2.	User/VO

WLCG Token Profile

A common set of claims is proposed for WLCG tokens, with additional claims specified for both Identity and Access tokens. Operational experience in the wider community indicates that performance and compatibility issues with existing libraries may be experienced if large tokens are used; this claims set has been developed with claim length minimisation as a priority.

WLCG Token Claims

This profile inherits from JSON Web Tokens at its base, including a specific claims language from RFC 7519. In this section, we outline the common WLCG-specific usage of the claims, for Access and ID Tokens, denoting any changes in claim criticality.

Common Claims

The following claims may be included in both Access and Identity tokens.

Claim	**Origin**	**Usage Note**	**Required[3]?**
sub	RFC7519 & OpenID Connect core	Typically indicates the individual or entity this token was originally issued to. The subject (**`sub`**) must be locally unique for a specific issuer, i.e. within the WLCG VO. It must be ASCII-encoded, not exceeding 255 characters in length, and is a case-sensitive string. Suggested use cases for the **`sub`** claim are suspending access to resources, auditing, user-level accounting, monitoring, or tracing. Due to privacy concerns, VOs SHOULD issue non-human-readable subjects. The **`sub`** MUST be unique and	Required

[3] Is the token issuer required to send this claim?

		non-reassigned within the VO. A VO MUST NOT use the same subject for multiple entities.	
exp	RFC7519 & OpenID Connect core	The interpretation for **`exp`** is unchanged from the RFC, it represents the expiration time on or after which the Token MUST NOT be accepted for processing.	Required
iss	RFC7519 & OpenID Connect core	The issuer (**`iss`**) of the WLCG JWT. It MUST contain a unique URL for the organization[4]; it is to be used in verification as described in the "Token Verification" section. For WLCG this would be the VO.	Required
wlcg.ver	WLCG AuthZ WG	We add the **`wlcg.ver`** claim to denote the version of the WLCG token profile the relying party must understand to validate the token (claim validation is covered in the next section). **`wlcg.ver`** names MUST comply with the following grammar: `vername ::= [0-9]+\.[0-9]+` The **`wlcg.ver`** claim corresponds to a version of this document. The initial version of this document constitutes version "1.0". Although versions are expected to be treated as strings, we adopt a numeric format for simplicity.	Required
eduperson_assurance	REFEDS	See below	Optional
acr	OpenID Connect core	The **acr** claim conveys the assurance of the authentication, e.g. Multi or Single Factor. It is typically included in addition to the eduperson_assurance claim.	Optional

[4] This implies running a token issuer in a high availability mode behind a single URL.

wlcg.groups	WLCG AuthZ WG	The **wlcg.groups** claim conveys group membership about an authenticated end-user. The claim value is an ordered JSON array of strings that contains the names of groups of which the user is a member in the context of the VO that issued the Token. Group names are formatted following the rules in the next section. Group names MUST comply with the following grammar[5] where group is defined recursively: `group ::= '/' groupname | group '/' groupname` `groupname :: = [a-zA-Z0-9][a-zA-Z0-9_.-]*` Usage of this claim is OPTIONAL. However, the **wlcg.groups** claim is REQUIRED in all tokens issued as a result of an OpenID Connect authentication flow in which wlcg.groups are requested via scopes and the subject is entitled to the groups in question. The group request mechanism is described in more detail in section “Scope-based Attribute Selection” of this document. *Note: it is expected that a more verbose syntax and different claim (eduperson_entitlement), as recommended by AARC[6] Guidelines, could also be required in the event that authorization information is exchanged with external Infrastructures.*	Optional, but when requested it MUST be present in both token types.
aud	RFC7519 & OpenID Connect core	The **aud** claim represents the audience or audiences the token is intended for. In the general case, the **aud** value is an array of case sensitive strings. In the common special case when there is one audience, the aud value MAY be a single case sensitive string. The special string value of “https://wlcg.cern.ch/jwt/v1/any” signifies that the issuer intends the token to be valid for all relying parties; this string value is required in	Required

[5] From GFD-I.182, the VOMS spec for FQANs (§3.4.1.4)
[6] https://aarc-project.eu/guidelines

		order to force an issuer to explicitly state the intent that the token is targeted to any relying party. The contents of the claim may either be a string or URL; we do not currently provide specific guidance on selecting audience names.	
iat	RFC7519	The claim represents the time at which the token was issued. Its value is a JSON number representing the number of seconds from 1970-01-01T0:0:0Z UTC until the token issue time in UTC.	Required
nbf	RFC7519	The interpretation for **`nbf`** (not before) is unchanged from the RFC. For example, usage of nbf allows the issuer to make the token valid prior to the issue instant, potentially easing clock skew issues in a distributed environment.	Optional
jti	RFC7519	The interpretation for **`jti`** (JWT ID) is unchanged from the RFC. It is a unique identifier that protects against replay attacks and improves traceability of tokens through the distributed system. It MUST be unique within an issuer and SHOULD be unique across issuers.	Required

ID Token Claims

For the ID Token schema we rely on the OpenID Connect (OIDC) standard, and in particular on the core specification. OpenID Connect is "a simple identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner." (for more information on OpenID Connect, refer to http://openid.net/connect/). We expect the validation of these tokens to additionally follow the corresponding flows in the OIDC standard (see ID token validation and code flow token validation).

OpenID Connect implements authentication as an extension to the OAuth 2.0 authorization process. Use of this extension is requested by Clients by including the `openid` scope value in the Authorization Request. Information about the authentication performed is returned in a JSON Web Token (JWT) often called an ID Token. The discussion on the OpenID Connect flows used to obtain the ID token is out of the scope of this document but referred to in the Appendix.

In the following section we describe the schema for identity related claims included in the ID token. Some of these claims MAY be also included in an access token, when the token is obtained through an OpenID Connect flow, or returned as the result of a call to the userinfo endpoint exposed by the OpenID Connect Provider issuing the token, or as the result of an access token introspection at the same provider.

The following additional claims are defined for WLCG ID Tokens. Other identity-related claims could be included in the ID Token, or returned in the result of calls to the userinfo or token introspection endpoint, following the recommendations of the OpenID Connect core profile.

Claim	Origin	Usage Note	Required?
auth_time	OpenID	The **auth_time** claim represents the time when the End-User authentication occurred. The claim value is a JSON number representing the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the End-User authentication time. As in the OpenID Connect core profile, the claim is REQUIRED when requested explicitly in the authentication request, otherwise is OPTIONAL.	Optional, but when requested it MUST be present
General OIDC Claims	OpenID	General OIDC claims may be included in tokens. For example, the **nonce**, **preferred_username**, and **email** claims that are derived from the OIDC core specification and follow the rules prescribed there.	Optional

Access Token Claims

The Access Token includes information about the authorization and rights the bearer is allowed to make use of. The Access Token is meant to be utilized on distributed services such as for storage or computing, whereas the ID Tokens is not intended to be sent to resource servers.

The Access Token profile contains two different approaches to authorization - group membership-based and capability-based, see the paragraph “Interpretation of Authorization by the Resource Server”.

When group membership is asserted, it is a statement that the bearer has the access privileges corresponding to the VO’s listed groups: it is up to the resource to determine the mapping of the group names to the access privileges. The technical profile of the group membership is described in the Common Claims section and not repeated here.

When a capability is asserted, it is relative to the VO's coarse-grained authorization; the resource only maps the token to a VO, then relies on the specified capability in the token for the fine-grained authorization within the VO's authorized area. In this way, the VO, not the resource, manages the authorizations within its area.

An access token SHOULD include at least the **`scope`** or **`wlcg.groups`** claim.

The following additional claims are defined for Access Tokens.

Claim	Origin	Usage Note	Required?
scope	Inspired by OAuth token exchange draft	See below	Optional

Claims defined by the WLCG Authorization Working Group should ideally be registered appropriately in the public domain[7].

Authorization

The token profile contains two different approaches to authorization - user attribute-based (e.g. groups or assurance) and capability-based.

Capability based Authorization: scope

Authorization may be based on the scope[8] claim. The value of the scope claim is a list of space-delimited, case-sensitive strings (as in OAuth Token Exchange draft 19, section 4.2) reflecting authorized activities the bearer of this token may perform.

We aim to define a common set of authorizations (particularly storage-related authorizations), but envision additional authorizations will be added to meet new use cases. The interpretation of such authorizations would result in a list of operations the bearer is allowed to perform.

For a given storage resource, the defined authorizations include:

[7] Such registrations could be made through IETF or appropriate bodies, and made publicly available e.g. https://www.iana.org/assignments/oauth-parameters/oauth-parameters.xml

[8] Note that the motivation of using the name "**scope**" here is inspired from the claim language proposed for standardization as part of the OAuth token exchange draft RFC, and due to its existing use in SciTokens.

- **storage.read**: Read data. Only applies to “online” resources such as disk (as opposed to “nearline” such as tape where the **stage** authorization should be used in addition).
- **storage.create**: Upload data. This includes renaming files if the destination file does not already exist. This capability includes the creation of directories and subdirectories at the specified path, and the creation of any non-existent directories required to create the path itself (note the server implementation MUST NOT automatically create directories for a client). This authorization DOES NOT permit overwriting or deletion of stored data. The driving use case for a separate `storage.create` scope is to enable stage-out of data from jobs on a worker node.
- **storage.modify**: Change data. This includes renaming files, creating new files, and writing data. This permission includes overwriting or replacing stored data in addition to deleting or truncating data. This is a strict superset of `storage.create`.
- **storage.stage**: Read the data, potentially causing data to be staged from a nearline resource to an online resource. This is a superset of `storage.read`.

For a given computing resource, the defined authorization activities include:

- **compute.read:** “Read” or query information about job status and attributes.
- **compute.modify:** Modify or change the attributes of an existing job.
- **compute.create:** Create or submit a new job at the computing resource.
- **compute.cancel:** Delete a job from the computing resource, potentially terminating a running job.

We use explicit "storage" and "compute" prefixes in the scope names in order to prevent token confusion at the issuer; if the unadorned string “upload” were used for both storage and compute cases, a token meant for uploading job results could potentially be usable for submitting jobs to a computing resource.

The operation definitions are currently kept open-ended and intended to be interpreted and evolved by the WLCG community.

Scopes MAY additionally provide a resource path, which further limits the authorization. These paths are provided in the form `$AUTHZ:$PATH`. For example, the scope `storage.read:/foo` would provide a read authorization for the resource at `/foo` but not `/bar`. Resources allow a hierarchical relationship to be expressed; an authorization for `storage.modify:/baz` implies a write authorization for the resources at `/baz/qux` (this is similar for `storage.read`) authorizations. Resources accepting scopes MUST handle these resource-based authorizations as described in this document; implementers should be aware this differs from the standard handling of OAuth2 scopes.

This authorization scheme is not equivalent to POSIX semantics. When mapping this authorization scheme to a POSIX-like filesystem, some considerations must be made for user and group ownership. For example, if a token is issued with authorization `storage.read:/home`, an implementation MUST override normal POSIX access control and give the bearer access to all users' home directories.

For all `storage.*` scopes, `$PATH` MUST be specified (but may be `/` to authorize the entire resource associated with the issuer); if not specified for these scopes, the token MUST be rejected. A token issuer MUST utilize absolute paths and normalize them according to section 6 of RFC 3986; as in RFC 3986, each component of the path must be URL-escaped. If a relying party encounters a non-conforming token, then it is implementation-defined if it rejects the token or performs path normalization.

The scope claim MAY include multiple authorizations of the same scope name, e.g. `storage.create:/foo storage.create:/bar`.

In the case of batch or computing resources, it is not clear how to define finer-grained resources. Currently, the authorizations of the relevant scopes (`compute.read, compute.modify, compute.create, compute.cancel`) refer to all jobs owned by the issuer. For example, a token with `compute.read` scope issued by https://cmsweb.cern.ch would be able to query the status of any CMS job at the resource.

When rendered in JSON, the value of the `scope` claim should be a space-separated list if there is more than one authorization present.

Examples values of the `scope` claim:

- `"storage.read:/"` This would allow a job (or any bearer) to read any file owned by the VO.
- `"storage.read:/protected storage.create:/protected/subdir"` This would allow a job to read the VO's data in the `/protected` subdirectory but (destructively) write into `/protected/subdir`.
- `"compute.create"` This would allow the bearer to submit jobs to a batch system on behalf of the issuing VO.
- `"storage.stage:/tape/subdir storage.read:/protected/data"` This would allow the bearer to read (and possibly stage) files in `/tape/subdir` and read files in `/protected/data`.
- `"storage.read:/store storage.create:/store/mc/datasetA"` This would allow the bearer to read from `/store` and create new files (not overwriting existing data) in `/store/mc/datasetA`.

Group Based Authorization: wlcg.groups

Authorization may be based on the `wlcg.groups` claim. The value of the `wlcg.groups` claim is an ordered JSON array of case-sensitive strings reflecting the VO groups of which the token subject is a member.

`wlcg.groups` semantics are equivalent to existing VOMS groups. VOMS roles should be considered as optional (i.e. returned only if requested) `wlcg.groups`; selection of optional groups is discussed below.

Interpretation of Authorization by the Resource Server

When groups are asserted (in an Access Token or ID Token, or both), it is a statement that the bearer has the access privileges corresponding to the VO's listed groups: it is up to the resource to determine the mapping of the group names to the access privileges.

When a capability is asserted (only in Access Tokens), it is relative to the VO's coarse-grained authorization; the resource only maps the token to a VO, then uses the capabilities in the token for fine-grained authorization within the VO's authorized area. In this way, the VO, not the resource, manages the authorizations within its area.

Access tokens may convey authorization information as both groups and capabilities. If both group membership and capabilities are asserted, then the resource server should grant the union of all authorizations for the groups and capabilities that it understands. The resource server may choose to not provide authorizations based on capabilities or may choose to not map the asserted groups to any authorization. Both assertions of group membership and capabilities are currently interpreted as positive authorizations.

Identity Assurance

The REFEDS Assurance Framework (RAF v1.0) splits assurance into three orthogonal components, namely, identifier uniqueness, identity assurance, and attribute assurance. For simplicity, RAF collapses the components into two assurance profiles Cappuccino and Espresso. AARC-G021 extends RAF with additional assurance profiles recommended to be used between infrastructures: IGTF-BIRCH, IGTF-DOGWOOD and a new specific profile addressing assurance derived from social-identity sources, AARC-Assam.

Since the assurance of authentication is not covered by RAF, the above profiles need to be used in conjunction with specifications focusing on authentication, such as the REFEDS SFA and REFEDS MFA profile. We adopt the `eduperson_assurance` multi-valued claim proposed by RAF[9] to convey the assurance component values and profile. The `acr` claim is included in

[9] RAF still refers to it as eduPersonAssurance, but it will probably change into `eduperson_assurance`, following the OIDCre whitepaper.

addition to the `eduperson_assurance` claim to specifically convey the authentication assurance.

In the case of this profile, identity assurance information will be sent by the WLCG token issuer whenever a user authentication flow is used to obtain the token.

Scope-based Attribute Selection

As defined in Section 3.3 of the OAuth 2.0 specification [RFC6749], "scopes" can be used to request that specific sets of information be made available as Claim Values. For WLCG, scopes are envisaged for requesting the inclusion of authorization information, returned as instances of the `wlcg.groups` claim and/or the `scope` claim (to convey capabilities). Scopes are also defined to request specific versions of the WLCG token schema.

Scope-based Group Selection

VOMS provides two main attribute types:

- **Groups**, which are used to assess group membership in the context of a VO.
- **Roles**, which are used to assess special privileges in the context of a VO or a specific group in the VO.

VOMS attributes are encoded to strings using a path-based syntax called **Fully Qualified Attribute Name** (FQAN), e.g.:

- /atlas/calib-muon, /cms/itcms (group FQANs)
- /atlas/Role=production, /cms/Role=pilot (role FQANs)

In VOMS, group membership is **always** asserted in an attribute certificate (AC) (i.e., the users get all the groups they belong to), while role inclusion is **optional**, and must be explicitly requested by the user.

VOMS also allows to impose an ordering on the requested attributes, since services mainly consider the first FQAN included in a VOMS AC (usually called the primary FQAN) for authorization.

We propose to use **scopes** to implement an attribute selection mechanism equivalent to the one provided by VOMS, following the approach outlined in the OpenID Connect standard:

- https://openid.net/specs/openid-connect-core-1_0.html#ScopeClaims

where scopes are defined and mapped to claims that are returned in access tokens, ID tokens and results for [userinfo endpoint]() and [token introspection]()requests.

In the proposed model, there are two types of groups:

- **Default groups**, whose membership is asserted regardless of explicit group membership requests
- **Optional groups**, whose membership is asserted only when explicitly requested by the client application

Default groups are similar to VOMS groups, while optional groups resemble VOMS roles.

A parametric wlcg.groups scope is introduced for group selection that has the following form:

```
wlcg.groups[:<group_name>]?
```

with the following rules:

- If the scope is parametric, i.e. it has the form wlcg.groups:<group_name>, the authorization server will return the requested group as a value in the wlcg.groups claim if the user is a member of the given group.
- To request multiple groups, multiple parametric wlcg.groups:<group_name> scopes are included in the authorization request.

- If the scope does not have the parametric part, i.e. its value is wlcg.groups, the authorization server will return the list of default groups (order is defined by the VO Administrator) for the user being authenticated for the target client. The default list of groups, including its order, is configurable by VO administrators, possibly even on a per-client basis.
- The order of the groups in the returned wlcg.groups claim complies with the order in which the wlcg.groups scopes were requested.

- If not explicitly included, the non-parametric wlcg.groups scope is implicitly added at the end of the requested scopes list whenever any group scopes are included. If no wlcg.groups scopes are included then it will not be added, to allow for cases where a client is only interested in capabilities.

- The returned wlcg.groups claim will not contain duplicates

If an entity is not entitled to a group, the scope requested may be ignored by the server and the corresponding token may not have the corresponding claims; in this case, section 3.3 of RFC 6749 requires the token issuer to inform the client. A server may also return an error during the authorization request. Client software implementations should always verify the scopes present in the returned token.

Examples:

In the following examples, “/cms” is the only default group.

Scope Request	Claim Result
`scope=wlcg.groups`	`"wlcg.groups": ["/cms"]`
`scope=wlcg.groups:/cms/uscms wlcg.groups:/cms/ALARM`	`"wlcg.groups": ["/cms/uscms","/cms/ALARM", "/cms"]`
`scope=wlcg.groups:/cms/uscms wlcg.groups:/cms/ALARM wlcg.groups`	`"wlcg.groups": ["/cms/uscms","/cms/ALARM", "/cms"]`
`scope=wlcg.groups wlcg.groups:/cms/uscms wlcg.groups:/cms/ALARM`	`"wlcg.groups": ["/cms", "/cms/uscms","/cms/ALARM"]`
`scope=wlcg.groups:/cms wlcg.groups:/cms/uscms wlcg.groups:/cms/ALARM`	`"wlcg.groups": ["/cms", "/cms/uscms","/cms/ALARM"]`

Scope-Based Capability Selection

Each desired capability should be requested in the scope request, following the recommendatIons of section 3.3 of RFC 6749.

If an entity is not entitled to a capability, the scope requested may be ignored by the server and the corresponding token may not have the corresponding claims; in this case, section 3.3 of RFC 6749 requires the token issuer to inform the client. A server may also return an error during the authorization request. Client software implementations should always verify the scopes present in the returned token.

Examples:

Scope Request	Claim Result
`scope=storage.read:/home/joe`	`"scope": "storage.read:/home/joe"`
`scope=storage.read:/home/joe storage.read:/home/bob`	`"scope": "storage.read:/home/joe storage.read:/home/bob"`
`scope=storage.create:/ storage.read:/home/bob`	`"scope": "storage.create:/ storage.read:/home/bob"`

Requesting Token Versions

To support future evolution of the WLCG token format, a client may add the requested token format as part of the scope request. A client wanting to receive a WLCG token should add the `wlcg` scope to its requests. If the client wants a specific version of a WLCG token, it should additionally append a `:` character and the version number (e.g., `wlcg:1.0` for a version 1.0 token).

For example, a client requesting a WLCG token with the compute.read scope would have the following scopes requested:

```
scope=wlcg compute.read
```

A client requesting a WLCG token formatted with version 1.0 and the /atlas/production group would have the following scopes requested:

```
scope=wlcg:1.0 wlcg.groups:/atlas/production
```

A server may decide to honor the client's token format and version request, ignore the request and issue a token with a different format or version, or return an error. A client SHOULD NOT assume the returned token has the requested version.

If no specific version is requested, the server may utilize a default version for issued token or it may associate a default version with the OAuth client's registration.

Security Considerations

Distribution of Trust

Within OAuth2 and OpenID Connect, clients need to fully trust the Authorization Servers (AS) or OpenID Connect providers (OP); in our model, these are under control of the VOs. At the same time, the issuers need to trust the clients to the point that they are willing to hand them a token on behalf of the end-users. Within the X.509 federation as used thus far, this distribution of trust was covered by the IGTF (Interoperable Global Trust Federation) and the e-Infrastructures distributing the set of trusted CAs; in the SAML world, this exchange of trust is handled by the different national federations and by eduGAIN on a global scale in the form of signed metadata exchange. On the other hand, OAuth2 and OIDC so far had very little use for a global trust federation, being used primarily by large social networks, whose business model presumes a single source of identity information (their own), and who typically allow any authenticated user to register new clients without further authorization, leveraging user consent to handle the trust and relying on the familiarity of the users with the limited number of OPs and ASes (everyone knows Google and Facebook). In the R&E context such a model is not workable: eduGAIN currently has close to 3000 IdPs and close to 2000 SPs requiring additional means of trust. One way is to require explicit approval of clients by the OP and AS operators, similar to what is done within (full mesh) SAML federations, but it was realized that such explicit approval will also not scale if the number of clients and OPs will start to grow. For OIDC, there is currently an effort to create an OIDC federation[10] which describes a way to distribute and delegate trust by forming 'federations' and 'sub-federations'. By leveraging the OIDC discovery[11] and OIDC dynamic registration[12] specifications this then provides a way of automatically obtaining client id and secret from OPs in the same OIDC federation.

For the WLCG, we foresee a limited number of registered OAuth2 clients - a small number per supported VO. This registration may be done via federation or out-of-band mechanisms; registration is not prescribed here. There will be a large number of unregistered resource servers that will need to verify the issued token; this verification is described in the next section. Additional features - web based federated login, token inspection or token exchange - will require registration, pragmatically limiting these features to the VOs.

[10] https://openid.net/specs/openid-connect-federation-1_0.html
[11] https://openid.net/specs/openid-connect-discovery-1_0.html
[12] https://openid.net/specs/openid-connect-registration-1_0.html

Example

A typical storage service must be able to map a token issuer (which corresponds to a single VO) to an area within the storage that the issuer is allowed to issue authorizations for. As an example, the XRootD implementation for JWT-based authorization has the following format:

```
[Issuer cms]
issuer = https://wlcg.example/cms
base_path = /users/cms


[Issuer dteam]
issuer = https://wlcg.example/dteam
base_path = /users/dteam
```

Here, the service administrator explicitly lists the issuers they trust (such as https://wlcg.example/dteam) and restricts each to a specific directory. The technical mechanism for verifying a token based on the trusted issuer name is given in the next section.

Token Verification

A token MUST be a properly-formatted JSON Web Token (JWT), as described by RFC 7519. In this subsection, we describe a mechanism to verify the token's authenticity in line with the standard.

The token MUST be signed with an asymmetric key (RSA- or EC-based signatures); the public key used to sign the token MUST be determined with the following algorithm.

- Extract the `iss` claim from the unverified token, check that the issuer is among the trusted ones and determine the JWKS URI using the approach described in the Metadata Lookup section in the Appendix.
- The contents of the JWKS URI MUST be compliant with RFC 7517. It provides a list of public keys associated with the issuer. The token MUST contain a key ID (`kid`) claim; the public key used to sign the token MUST be identified by matching the token's `kid` claim with the corresponding key ID in the JWKS key set.
- Once the public key is determined, the verification of the token and its signature can proceed as outlined in RFC 7519.

All communication between the resource and the issuer MUST be done over a valid HTTPS connection with hostname verification. The token issuer SHOULD advertise the public key lifetime by setting the appropriate HTTP caching headers. The Client SHOULD use HTTP headers to avoid unnecessary downloads. The recommended lifetime of the public key cache is one day, but SHOULD be kept to less than 7 days. Client implementations SHOULD cache the public key for an authorization server for at least 1 hour, regardless of the server-provided value. Reducing the lifetime of a key will likely impact network traffic.

Metadata lookup

All token issuers for the WLCG MUST follow the rules defined in the OpenID Connect discovery standard, i.e. provide the server metadata at the `.well-known/openid-configuration` sub-resource[13].

That is, if the issuer is https://dteam.wlcg.example, then the server metadata must be available at https://dteam.wlcg.example/.well-known/openid-configuration . See the OAuth 2.0 Authorization Server Metadata document (RFC8414) for a discussion on handling issuers with sub-paths, such as https://wlcg.example/dteam; it notes that https://wlcg.example/.well-known/openid-configuration/dteam is preferred but https://wlcg.example/dteam/.well-known/openid-configuration is acceptable as a fallback for existing clients. Further, the JWKS URI key MUST be provided within this configuration file.

The token issuer endpoint is a crucial point of trust between the service and the VO; hence, the TLS connection MUST be validated and verified according to best practices. The trust roots will be needed by a wide variety of agents, including browser-based and terminal-based clients[14].

Signature algorithms are enumerated in RFC 7518 section 3. The HMAC algorithms are incompatible with the WLCG JWT approach; implementations should use the recommended algorithms from the RFC (as of July 2018, this is ES256 or RS256; ES256 should be used when token-length is a concern). Changes to the allowable signature algorithms will be handled using the versioning mechanism described in the Token Validation section.

Verification Example

The RP needs to get hold of the https://dteam.wlcg.example issuer's keys for remote verification (which is necessary for scalability). For verification, a *minimal* OIDC discovery configuration file would be:

```
{
   "issuer":"https://dteam.wlcg.example",
   "jwks_uri":"https://dteam.wlcg.example/oauth2/certs",
}
```

[13] Note that the OpenID Connect Discovery paper highlights a mechanism that is NOT RFC 5785 compliant and is not aligned with the OAuth discovery standard. After some discussion this group decided to embrace the OpenID Connect Discovery approach. More details in the appendix.

[14] Each OS platform has its own set of acceptable CAs; suitable certificates should be used to facilitate client development and maintain the existing level of trust. Discussions will be held between the WLCG Authorization Working Group, IGTF and relevant partners.

For a usable OAuth2-based system, the `token_endpoint` would also need to be provided and a mechanism for how OAuth2 clients could register with the issuer; those are out of scope for this document.

The contents of the JWKS URI contains the VO's signing keys; an example:


```
{
    "keys": [
        {
            "alg": "RS256",
            "e": "AQAB",
            "kid": "key1",
            "kty": "RSA",
            "n":
"oj5UxvZGgQU2UHGdO2ViR6zkilHjTSFdTA_Jtb1KPKmqr3I7W-5YqI3xrIJoYNeazXGA
8l0w89BWfbet3NY8rlEocupmLpmeSRTh29DAIIskVKBevr2QbF-9qwunaLoMpal2ZFJTk
bMweiFiq-duhzcKI1JuaNkUJJpd6BGXVoszn31KHlVkUxd739FYyKLArUnnLRzQ6Ld6VD
iJrhRLnkUdXgitJuCy0gPaky9dWIVcKnjCI6F7F2o77II1m5lk3J9g6Dn6rfT6QppBQPz
_7t1LN-PIs-lO50nEsiDPHhb7GI0XucajA9ZAGXIPR11okFZqRuUaVnxizNXb1rHPmQ==
",
            "use": "sig"
        }
    ]
}
```


So, given a token header and payload (shown here not base64 encoded for human readability purposes):


```
{"alg":"RS256","typ":"JWT","kid":"key1"}
{"jti":"40ce5a87-e419-4bdf-9e11-61dfb160f89d","sub":"e1eb758b-b73c-47
61-bfff-adc793da409c","exp":1522064875,"iss":"https://dteam.wlcg.exam
ple","iat":1522057675,"scope":["read:/store","write:/store/user/arese
archer"],"nbf":1522057675,”wlcg.ver”:”1.0”}
```


One would utilize the `iss` claim in the payload to download the set of public keys from JWKS URI, then utilize the `kid` claim in the header to discover the public key used to sign the JWT.

Token Validation

Token Lifetime Guidance

Token Type	**Recommended Lifetime**	**Minimum Lifetime**	**Maximum Lifetime**	**Justification**

Access Token & ID Token[15]	20 minutes	5 minutes	6 hours	Access token lifetime should be short as we do not foresee the deployment of a revocation mechanism. The granted lifetime has implications for the maximum allowable downtime of the Access Token server.
Refresh Token	10 days	1 day	30 days	Refresh token lifetimes should be kept bounded, but can be longer-lived as they are revocable. Meant to be long-lived enough to be on a "human timescale." Refresh tokens are not necessarily signed and not tied to issuer public key lifetime.
Issuer Public Key Cache	6 hours	1 hour	1 day	The public key cache lifetime defines the minimum revocation time of the public key. The actual lifetime is the maximum allowable downtime of the public key server.
Issuer Public Key	6 months	2 days	12 months	JWT has built-in mechanisms for key rotation; these do not need to live as long as CAs. This may evolve following operational experience, provision should be made for flexible lifetimes.

Note the combination of **nbf** (not before) (or **iat**) and **exp** (expiration) provides a notion of token valid lifetime. WLCG token issuers MUST issue Access tokens with valid lifetime of less than 6 hours; they SHOULD aim for a token lifetime of 20 minutes. Resource providers MUST NOT

[15] It is not required that the two token lifetimes be identical. Typically Access Tokens are longer lived than ID Tokens.

accept tokens that have validity longer than 6 hours. As a pragmatic guard against minor clock skews, they SHOULD accept expired tokens that are less than 60 seconds expired. See the recommendations in sections 5.3 and 5.2 in RFC 6750. These tokens are purposely shorter-lived as they do not have a token revocation mechanism; the token lifetime should be shorter than the expected revocation response time for authorizations.

Refresh tokens and token revocation

Refresh tokens are credentials that can be used by client applications to obtain new access or ID tokens (when such tokens are about to expire) or to obtain access tokens with identical or narrower scope from an OAuth Authorization Server. Unlike access tokens, refresh tokens are intended for use only with authorization servers and are never sent to resource servers. Furthermore, the client needs to authenticate at the Authorization Server using its client credentials when using its refresh tokens.

Refresh tokens are typically longer lived than access tokens, and are used in support of long-running computational activities that last longer than the lifetime of a single access token.

As clarified in the OAuth specification, 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. Delegation across services in support of long-running jobs MUST leverage the token exchange flow. Refresh tokens SHOULD be kept on centrally maintained (non-grid) services while long-running jobs SHOULD get only access tokens[16]. Grid jobs SHOULD NOT be OAuth clients.

In order to contain security incidents related to the leakage of refresh tokens, it is recommended that any solution that will be used as the WLCG OAuth authorization server MUST support the OAuth token revocation standard (RFC 7009) at least for refresh tokens.

Claim and Token validation

The claims in a WLCG token are meant to indicate an identity or manage access to a resource. For example, in the authorization schema, additional claims might add restrictions to the corresponding bearer's authorizations: if an unknown claim is skipped, the resource provider may inadvertently offer overly-broad authorizations. On the other hand, requiring *all* claims to be processed may reduce the flexibility and ability to add future features.

To handle this challenge, each token MUST provide a **`wlcg.ver`** (version) attribute, whose value corresponds to an enumerated set of claims described earlier in this document. For that version of the token format, the corresponding claims MUST be handled by the implementation. Any additional claim present MUST be ignored (for access tokens, these claims MUST NOT be used in authorization decisions).

---

[16] The WLCG Authorization Working Group plans to produce guidelines for implementors on this workflow.

Each client library implementation MUST know the versions it supports; if it encounters a token whose **`wlcg.ver`** value is not supported by the implementation, the token MUST be rejected as invalid.

Additionally, signature algorithms and RS256, ES256 MUST be supported.

Operational Impact of Verification and Refresh

For operational stability and scalability it would be desirable to put reasonable constraints on the frequency at which token issuing services need to be contacted by the vast majority of relevant workflows.

By design, access tokens should be issued not long before they are used and hence can have short lifetimes. As a consequence, token issuers may already experience high request rates for issuing tokens alone. The usage of tokens in ALICE grid workflows has demonstrated the feasibility of such services on the scale of the LHC experiments, though some consideration should be given to corresponding requirements on the experiment services that provide such functionality. It would be desirable not to add yet more load on those services for other reasons. For the verification of an access token there would ideally be no need to contact the issuer at all, as is currently the case for VOMS proxies.

Access tokens are signed by keys with a lifetime for which a relatively short upper limit, as defined above, is deemed desirable. Each grid service supporting such tokens will regularly need to query each issuer for its set of currently valid public keys. For example, it might do that a few times per day and cache the results, as is currently done for CRLs. Such functionality may be similarly provided by an independent utility invoked by `cron`.

As access tokens typically will have short lifetimes of the order of 1 hour, there is no need to implement any revocation for them, whereas a revoked public key would simply no longer be served by the issuer.

As refresh tokens are longer-lived, different considerations apply to them. First, their maximum lifetime may need to be able to bridge the many hours that a pilot job may spend in a batch queue before it is able to start an actual user payload for which a fresh access token needs to be obtained. Similarly, a file transfer request may need to wait for many hours or even days in the queue of an FTS instance. Furthermore, certain payload jobs may need to run for many hours before their output can be uploaded. Today, the longest payloads need at least the better part of a day, possibly more. It would thus seem desirable to allow refresh tokens to have a maximum lifetime of at least 1 day. It would also be desirable to have the maximum not much higher, to limit the amount of damage that could be inflicted by a third party that came into possession of such a token. As refresh tokens are only used to obtain fresh access tokens from the original issuer, only the latter is concerned with refresh token revocation. The issuer is expected to provide a service endpoint where any such token can be revoked by its owner. It might be desirable for standard workflows to revoke a refresh token as soon as it is deemed to be no longer needed, though that would add to the request rates experienced by the issuer.

Appendix

Discovery

What is Discovery (the metadata lookup process)?

For the OpenID connect discovery standard it is “a mechanism for an OpenID Connect Relying Party to discover the End-User's OpenID Provider and obtain information needed to interact with it, including its OAuth 2.0 endpoint locations.”

For the OAuth authorization server metadata standard it is “a metadata format that an OAuth 2.0 client can use to obtain the information needed to interact with an OAuth 2.0 authorization server, including its endpoint locations and authorization server capabilities.”.

Well-known URIs

According to RFC 5785, a well-known URI is a URI whose path component begins with the characters “/.well-known/”, and whose scheme is "HTTP", "HTTPS", or another scheme that has explicitly been specified to use well-known URIs.

The OpenID connect approach to well-known URIs

The OpenID Connect discovery mechanism states that the well-known URI for an OpenID Connect provider is computed as follows (assuming the client knows the Issuer string of such provider):

1. if the Issuer does not contain any path component, the openid-configuration is resolved by querying the “/.well-known/openid-configuration” endpoint. Example: for https://wlcg.example the configuration URI would be https://wlcg.example/.well-known/openid-configuration
2. if the Issuer contains a path component, the “/.well-known/openid-configuration” path is appended to the Issuer string after having removed any terminating “/” character. Example: for https://wlcg.example/dteam the configuration URI would be https://wlcg.example/dteam/.well-known/openid-configuration

As clarified here, “using path components enables supporting multiple issuers per host. This is required in some multi-tenant hosting configurations. This use of .well-known is for supporting multiple issuers per host; unlike its use in RFC 5785, it does not provide general information about the host."

The OAuth approach to well-known URIs

The [OAuth authorization server metadata standard]() states that:
"Authorization servers supporting metadata MUST make a JSON document containing metadata as specified in Section 2 available at a path formed by inserting a well-known URI string into the authorization server's issuer identifier between the host component and the path component, if any. By default, the well-known URI string used is "/.well-known/oauth-authorization-server".

The OAuth approach is equivalent to the one standardized in OpenID connect discovery when the Issuer URI does not contain path components. However the two standards differ when a path component is present, since OpenID connect states that the well-known URI string is appended to the issuer string (https://example.com/issuer1/.well-known/openid-configuration) while OAuth states that the well-known URI should be inserted before the path component (https://example.com/.well-known/openid-configuration/issuer1).

The OAuth discovery standard states also that "when deployed in legacy environments in which the OpenID Connect Discovery 1.0 transformation is already used, it may be necessary during a transition period to publish metadata for issuer identifiers containing a path component at both locations. During this transition period, applications should first apply the transformation defined in this specification and attempt to retrieve the authorization server metadata from the resulting location; only if the retrieval from that location fails should they fall back to attempting to retrieve it from the alternate location obtained using the transformation defined by OpenID Connect Discovery 1.0."

Based on the rules above, the OAuth and OpenID connect discovery standards are aligned for single tenant OpenID Connect providers (assuming that path fragments are included in the Issuer string only to support multi-tenant OpenID Connect provider deployments).

For multi-tenant OpenID Connect providers, WLCG AuthZ WG should recommend that all providers and clients MUST support the OpenID connect discovery approach (given it is already widely implemented and is the standardized approach for OpenID connect (RFC8414)). Provider and client software MAY in addition also support the OAuth discovery approach described above.

OpenID Connect/OAuth authentication and authorization flows for WLCG

This appendix provides a brief introduction for the main authentication/authorization flows provided by OpenID Connect and OAuth that will be used by WLCG services to integrate token-based authentication and authorization.

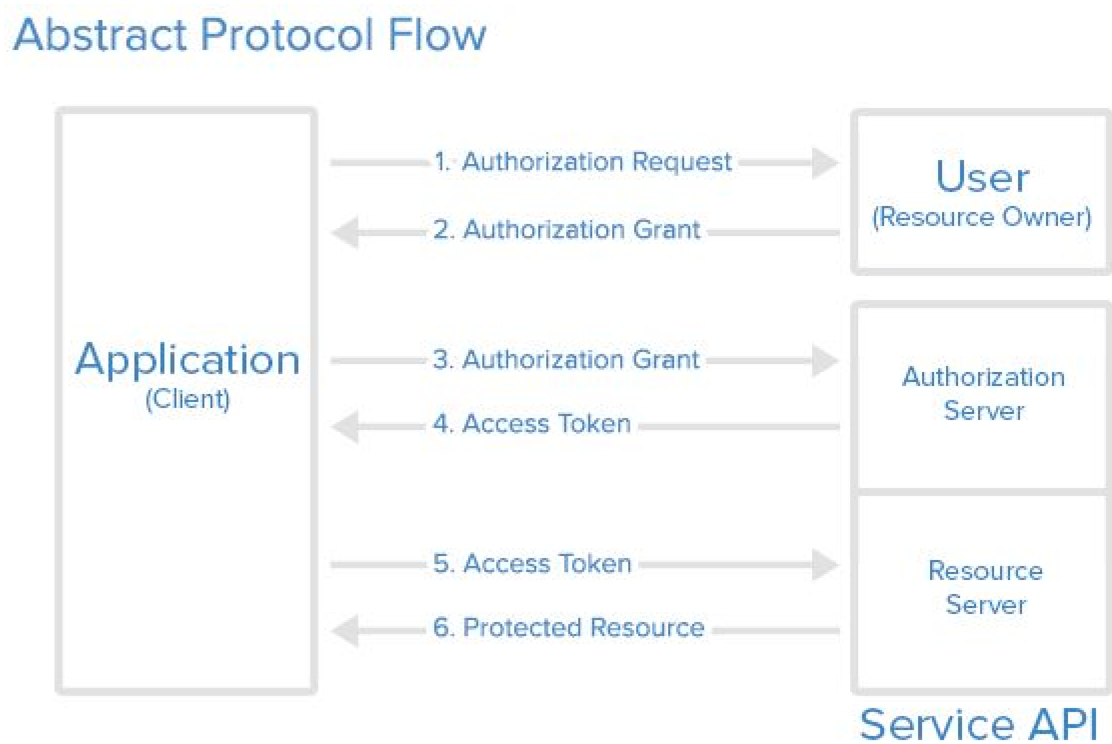
Confidential vs public clients

The OAuth specification defines two client application types (confidential and public) based on their ability to authenticate securely with the authorization server:

- Confidential client applications (e.g., server-side applications) are capable of maintaining the confidentiality of their credentials, or are capable of secure client authentication through other means.
- Public client applications (e.g., Javascript single page or mobile applications running on the user device) are by design incapable of storing client credentials in a secure way or do not have other means that allow for secure client authentication.

Some authorization flows or capabilities are available only to confidential clients. Most WLCG services can be classified as confidential clients.

Authorization flows



The image above, taken from here, describes the abstract OAuth protocol flow. In OAuth terminology, authorization flows describe the interactions among the roles defined above (a client application, the user, the authorization and resource servers) to let a client application obtain controlled access to protected resources with the (possibly implicit) authorization of the end user owning such resources. OAuth/OpenID connect flows can be described as variations of the abstract protocol flow in support of specific authentication and authorization scenarios.

OAuth is about delegating access to resources to third-party applications. This delegation process starts with an authorization request (step 1. in the figure above) issued by the application that wants to access the resources to the user owning the resources to get an “authorization grant”, i.e. a permission. Note that in some flows and under certain conditions (e.g., a trusted client application, a previous user authorization stored in the authorization server) the grant can be “implicit”, i.e. no explicit user intervention is required. The application then exchanges the grant obtained from the user with an access token. This access token is then presented to the resource server in order to get access to resources. The resource server will validate the token and grant access to the requested resources only if the token presented by the client application is valid and provides enough privileges to access the requested resources.

Authorization code flow

The authorization code flow is defined in RFC 6749 and extended in the OpenID Connect core specification. This flow is used to obtain access tokens, ID tokens and refresh tokens and is optimized for confidential clients (i.e., server-side applications).
In WLCG, we require the use of the OpenID Connect version of the code flow, as described here, which in practice means that the “openid” scope must be always included in authorization requests.

Refresh token flow

The refresh token flow is also targeted at confidential clients and is used to obtain new access tokens when tokens are expired or about to expire. This flow does not require the user presence, and is mainly used to support offline activities, when a client application needs to act on behalf of a user for a possibly unbounded amount of time.
Refresh tokens can be obtained using any flow that supports them (e.g., the authorization code flow), typically by including the recommended “offline_access” scope in authorization requests.

Device flow

The device flow is an authorization flow developed in support of devices with limited input capabilities. In WLCG, we will use this flow mainly to support command line interface (CLI) applications, while preserving the flexibility of using a browser for the user authentication flow.

Client credentials flow

Sometimes client applications need to interact with services in a way that is not bounded to any specific user, but with the client application itself. In support of this use case OAuth provides the client credentials flow.

Token exchange flow

The OAuth token exchange flow can be used to implement delegation and token privileges attenuation across a chain of services.

Examples

Device flow token request example

The OAuth device code flow enables OAuth on devices that have internet connectivity but lack a browser or an easy way to enter text. In this flow, the device instructs the user to open a URL on a secondary device such as a smartphone or computer in order to complete the authorization. There is no communication channel required between the user's two devices.
It is convenient of our CLI use cases since it enables federated authentication from a terminal (assuming the user has access to a browser, which is the case for most of our use cases). The authorization flow is triggered by a registered client application with an HTTP POST request at the Authorization Server device code endpoint:

```
POST /devicecode HTTP/2
Host: iam-escape.cloud.cnaf.infn.it
Authorization: Basic ZG9….EwxZnFsX2lWZmlSamR
User-Agent: curl/7.65.3
Accept: */*
Content-Length: 61
Content-Type: application/x-www-form-urlencoded

client_id=doma-test&scope=openid profile email offline_access
```

The authorization server authenticates the clients, and returns a user code, a device code and a URL:

```
{
  "user_code": "41SGWX",
  "device_code": "da317e13-d881-4980-ad33-7f4db7169930",
  "verification_uri": "https://iam-escape.cloud.cnaf.infn.it/device",
  "expires_in": 1800
}
```

The user code and a URL are shown by the script to the user in a more palatable way:

```
Please open the following URL in the browser:

https://iam-escape.cloud.cnaf.infn.it/device

and, after having been authenticated, enter the following code when
requested:
```

```
41SGWX

Note that the code above expires in 1800 seconds...
Once you have correctly authenticated and authorized this device,
this script can be restarted to obtain a token.

Proceed? [Y/N] (CTRL-c to abort)
```

The user can then authenticate with his browser and grant access to the device client. Once the authentication flow on the browser is complete, the user comes back to her terminal and types Y to proceed.
The script then submits the following HTTP request:

```
POST /token HTTP/2
Host: iam-escape.cloud.cnaf.infn.it
Authorization: Basic ZG9….EwxZnFsX2lWZmlSamR
User-Agent: curl/7.65.3
Accept: */*
Content-Length: 104
Content-Type: application/x-www-form-urlencoded

grant_type=urn:ietf:params:oauth:grant-type:device_code&device_code=d
a317e13-d881-4980-ad33-7f4db7169930
```

And gets the back the requested tokens:


```
{
  "access_token":
"eyJraWQiOiJyc2ExIiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiIyNDE2ODdlOC01Mzc0L
TQ1NDktYjlmNi1hMzg2NmYwYmY2ZGEiLCJzY29wZSI6ImVtYWlsIG9wZW5pZCBvZmZsaW
5lX2FjY2VzcyBwcm9maWxlIiwiaXNzIjoiaHR0cHM6XC9cL2lhbS1lc2NhcGUuY2xvdWQ
uY25hZi5pbmZuLml0XC8iLCJleHAiOjE1Njc3ODI2MzAsImlhdCI6MTU2Nzc3OTAzMCwi
anRpIjoiN2JhMDdlY2YtY2NkOS00MTA3LWJkMzMtOWRlNWNjOGJkMjU5In0.KRlCP26GI
qxcgK-B43j8cIQP8DSape9ekTyY46P6ojGQnfjLdZRS3sOvyWM6SGDRbsMdjfIr96iExt
2yj-82nVOiv6yq2MIxgmLI1Ue_gQYnI-D9fh75b6z-S6FhMwsbsE1Ox9s1BthPkbbofYV
wPCLNFKKga39qqSe-urUMN9E",
  "token_type": "Bearer",
  "refresh_token":
"eyJhbGciOiJub25lIn0.eyJqdGkiOiJmZTU0MGMzMi0xMDZlLTRhMDQtYjMyYy05MGNh
OTBjNzY2MDUifQ.",
  "expires_in": 3599,
  "scope": "email openid offline_access profile",
```

```
  "id_token":
"eyJraWQiOiJyc2ExIiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiIyNDE2ODdlOC01Mzc0L
TQ1NDktYjlmNi1hMzg2NmYwYmY2ZGEiLCJhdWQiOiJkb21hLXRlc3QiLCJraWQiOiJyc2
ExIiwiaXNzIjoiaHR0cHM6XC9cL2lhbS1lc2NhcGUuY2xvdWQuY25hZi5pbmZuLml0XC8
iLCJuYW1lIjoiQW5kcmVhIENlY2NhbnRpIiwiZ3JvdXBzIjpbImVzY2FwZSIsImVzY2Fw
ZVwvcGlsb3RzIiwiZXNjYXBlXC94ZmVycyJdLCJwcmVmZXJyZWRfdXNlcm5hbWUiOiJhb
mRyZWEiLCJvcmdhbmlzYXRpb25fbmFtZSI6ImVzY2FwZSIsImV4cCI6MTU2Nzc3OTYzMC
wiaWF0IjoxNTY3Nzc5MDMwLCJqdGkiOiJkNDExNGEyNS0wZDA4LTRiNWEtYmQ1Zi00YzI
zYTM2NmZhODciLCJlbWFpbCI6ImFuZHJlYS5jZWNjYW50aUBnbWFpbC5jb20ifQ.XUjVH
QmBel1_W_7qlF62ysQKnjSMHSGtGTzVmNPKZkzuEiI3wGZiwJussPWkhC3-TiNtO--_G3
3kre6gXqJusYLEH-LUPYVcfBZ85KLPmmG17EE10XmKBFg3-eyonstaRAHoXwM4YHvMnA5
tgtkawZsry4ssSA2WZz7BKBhffWQ"
}
```

OAuth2 Auto-Discovery and Token Request

Suppose an end-user would like to request a token from the issuer at https://cmsweb.cern.ch. First, auto-discovery must be performed at the endpoint to determine the token issuer:

```
> GET /.well-known/openid-configuration HTTP/1.1
> User-Agent: curl/7.29.0
> Host: cmsweb.cern.ch
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Content-Length: 155
{
   "issuer":"https://cmsweb.cern.ch",
   "jwks_uri":"https://cmsweb.cern.ch//oauth2/certs",
   "token_endpoint":"https://cmsweb.cern.ch/oauth2/token"
}
```

From the JSON response, the endpoint the client must contact is at https://cmsweb.cern.ch/oauth2/token. Next, the client may perform an OAuth2 request against this endpoint. In the case where the client has the ability to perform a traditional (for the WLCG community) X509-client certificate authentication, it may connect to this endpoint and perform a client credentials flow, utilizing the X509 client authentication as the implicit client credentials. The HTTP request/response would look as follows:

```
> POST /oauth2/token HTTP/1.1
> User-Agent: curl/7.29.0
> Host: cmsweb.cern.ch
> Accept: */*
```

```
> Content-Length: 65
> Content-Type: application/x-www-form-urlencoded
>
grant_type=client_credentials&expire_in=3600&scope=storage.read:/home
/joe
```

```
< HTTP/1.1 200 OK
< Content-Length: 374
< Content-Type: application/json
<
{"access_token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6ImtleS1yczI1NiJ9.eyJpYXQ
iOjE1NTUwNjAxMjAsInN1YiI6ImpvZSIsImV4cCI6MTU1NTA2MDcyMCwiaXNzIjoiaHR0
cHM6Ly9kZW1vLnNjaXRva2Vucy5vcmciLCJqdGkiOiJhNDZhZTk5MS02ZjFjLTRlMDYtO
Tc5Yi05Njc5NjY3NDBhYmIiLCJ3bGNnLnZlciI6IjEuMCIsIm5iZiI6MTU1NTA2MDEyMC
wic2NvcGUiOiJzdG9yYWdlLnJlYWQ6L2hvbWUvam9lIn0.VOxWM_PV3aikQK875xs1Aab
Bmk3VvMLJZ9Rrp08juN9Jux-NMUwHQGANA3-dd4_IWGLYtmCn4lyr4aiPuGCRk5D35Or7
W1TjSTHmD116iuH3_U_WPgE5-TOGHnAYoE8nsdVUQ4UOKj64DqPOXABdBsgsqRML-MpYi
S5p6K2yhcIdPAk-M5kXYVWu2vr8A-VLUNXwM7oYoe9mJ_9_rYCZs-aWqZ-x22AmnoTtTV
Kx5Wur4cSmL308CBr-yltZf_OtM4eIjjrDFbioIjbl-hG-y51oJU0n7n07-hxHeqWXO6R
Wpo_fUWwrrlrm8FRJXbk_8d33bdhodQ5w8DwEbF79fQ", "token_type": "bearer",
"expires_in": 3600}
```

In this example, the payload has the following decoded contents:

```
{
  "wlcg.ver": "1.0",
  "jti": "a46ae991-6f1c-4e06-979b-967966740abb",
  "iss": "https://demo.scitokens.org",
  "sub": "joe",
  "nbf": 1555060120,
  "iat": 1555060120,
  "exp": 1555060720,
  "scope": "storage.read:/home/joe"
}
```

Note this is a very minimal token for this JWT profile, contains no group information, and does not use a de-identified subject claim.

Example Identity Token

```
{
     "wlcg.ver": "1.0",
     "sub": "e1eb758b-b73c-4761-bfff-adc793da409c",
     "iss": "https://dteam.wlcg.example",
```

```
      "wlcg.groups": [
        "/dteam/VO-Admin",
        "/dteam",
        "/dteam/itcms"
      ],
      "preferred_username": "aresearcher",
      "nonce": "334b0e05b65a3",
      "aud": "https:///dteam-test-client.example.com",
      "auth_time": 1523363636,
      "name": "A Researcher",
      "exp": 1523365436,
      "iat": 1523363636,
      "jti": "aef94c8c-0fea-490f-9027-ff444dd66d8c",
      "email": "a.researcher@cern.ch",
      "eduperson_assurance" : [
        "https://refeds.org/assurance/profile/espresso"
      ],
      "acr": "https://refeds.org/profile/mfa"
}
```


In this example, the `nonce`, `preferred_username`, `name` and `email` claims are derived from the OIDC core specification and follow the rules prescribed there. For example, OIDC specification states the relying party should NOT treat `preferred_username` as a unique identifier; this is also true in the WLCG profile.

Example Access Token with Groups


```
{
      "sub": "e1eb758b-b73c-4761-bfff-adc793da409c",
      "iss": "https://demo.scitokens.org",
      "nbf": 1555059791,
      "wlcg.ver": "1.0",
      "aud": "https://dteam-test-client.example.com",
      "exp": 1555060391,
      "iat": 1555059791,
      "jti": "aef94c8c-0fea-490f-9027-ff444dd66d8c",
      "wlcg.groups": [
        "/dteam/VO-Admin",
        "/dteam",
        "/dteam/itdteam"
      ],
      "eduperson_assurance": [
        "https://refeds.org/assurance/profile/espresso"
      ],
      "acr": "https://refeds.org/profile/mfa"
}
```

Example Access Token with Authorization Scopes


```
{
      "sub": "e1eb758b-b73c-4761-bfff-adc793da409c",
      "iss": "https://demo.scitokens.org",
      "nbf": 1555059791,
      "wlcg.ver": "1.0",
      "aud": "https://dteam-test-client.example.com",
      "exp": 1555060391,
      "iat": 1555059791,
      "jti": "aef94c8c-0fea-490f-9027-ff444dd66d8c",
      "scope": "storage.read:/store storage.create:/store/mc/datasetA
      compute.create:/",
      "eduperson_assurance": [
        "https://refeds.org/assurance/profile/espresso"
      ],
      "acr": "https://refeds.org/profile/mfa"
}
```