JWT Shared Profile for WLCG

Authors: *M Martinez Pedreira, M Litmaath, P Millar, A Ceccanti, M Sallé, B Bockelman, H Short, N Liampotis*

This document describes the current usage of authorization tokens in projects of interest to the WLCG community. From here we aim to develop a shared JWT profile that is interoperable across infrastructures participating in WLCG. The proposed profile should be widely circulated and undergo a period of community consultation.

Scope

To define a schema for an authorization token to be used by participating infrastructures.

Example Use Cases

Please describe how you are using authorization tokens. Free form text and we can neaten later as needed. Include details of claims used.

ALICE (Miguel)

1. Token Certificates (for services authentication and authorization)

In the new Java-based ALICE grid framework, JAliEn, the base set of credentials for an ALICE user or entity are the X509 user certificates. In order to interact with the VO services, the user can keep using the standard user certificates or generate a Token Certificate of a certain type and characteristics. This is especially useful when we want to delegate it to another component, like a pilot, or a payload. The types of tokens a user is allowed to request are based on a central role-based configuration. So each user can be mapped to specific roles like ‘vobox’ or ‘pilot’. Then, each token can only execute a fixed set of operations, which are controlled by the central services and checked and routed on every request. For example, the ‘jobagent’ Token Certificate can only do job matching operations, and is only attached to pilot submitter users.

The nature of the Token Certificates is based on X509 as well. In fact, they are full-fledged X509 certificates, with special properties encoded on them. We re-purpose the DN and extensions on it to map this information, while we also explored more options. This makes the software easily compatible with the current X509 security libraries while adding the flexibility of mapping roles and operations to the tokens, as well as getting rid of the limitations and lack of future support of proxies.
All that is needed to validate the calls with tokens are the CAs which are already distributed and necessary everywhere.

A utility has been developed as well to renew tokens automatically if desired. This is interesting for users not having to re-type passwords during their sessions, or for long-lived services on the VoBoxes, among others.

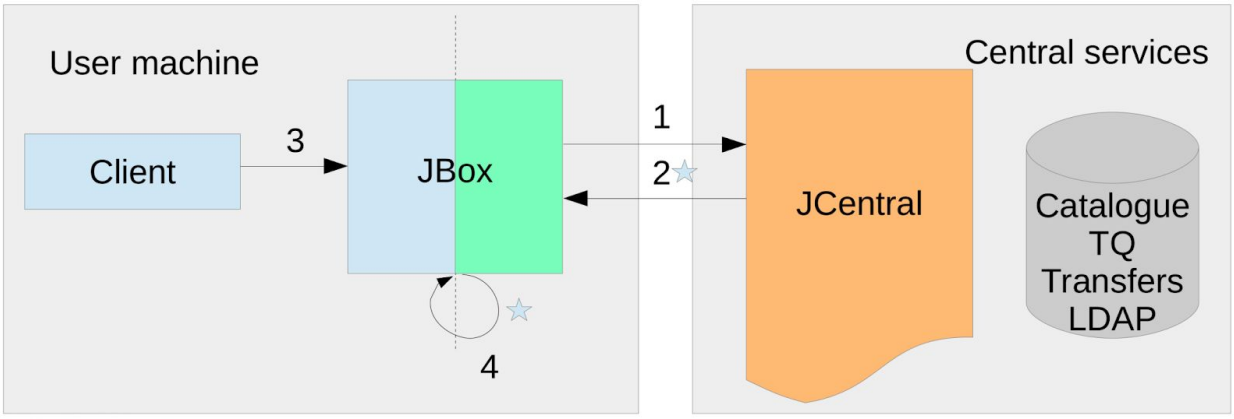
An example of a Token Certificate for a payload wrapper (which is a user temporary certificate linked to a unique job execution) would be:

Subject: OU="queueid=1038905674/resubmission=3/user=mmmartin", OU=jobagent, CN=jobagent, CN=Jobs, O=AliEn, C=ch- Issuer: JAliEn-CS

This means the agent can operate on job 1038905674 as user 'mmmartin' with the full rights of this user. The resubmission field let us know if this a correct execution of such a payload (when we resubmit a payload we want to make sure the old execution is not able to authenticate anymore, so it doesn't produce conflicts).

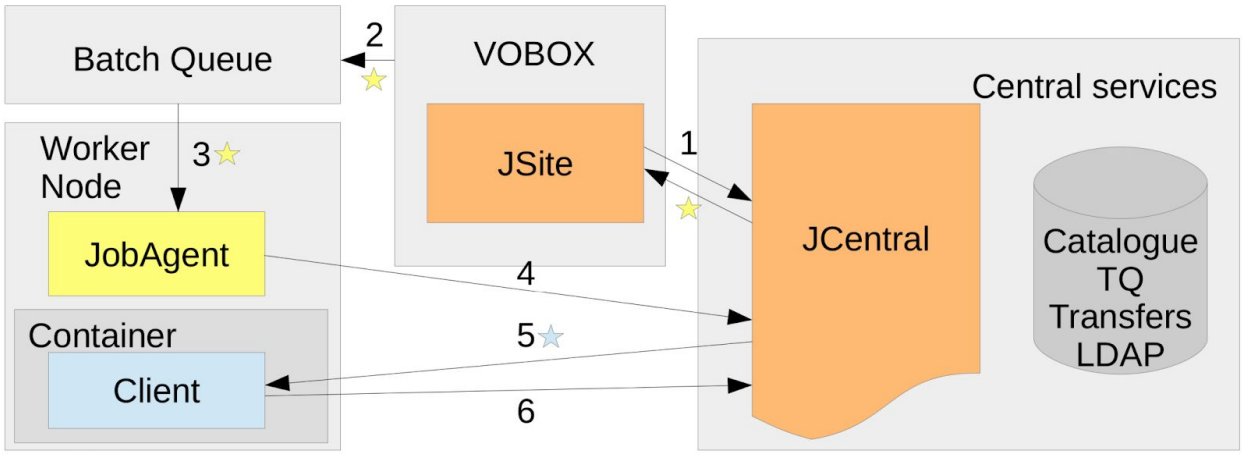
As a final note, mind that these tokens are implemented and tested, but we still don't have sites in production using this schema. It will happen early 2018, in principle, when we complete the full re-implementation of the new grid framework.

Long living authentication (agent) for client



1. Initial connect with usercert
2. Get tokencert
3. Client always uses tokencert
4. Renew automatically every 2 hours

Pilot and Job auth and execution



1. Request new job agent token certificate every 5 minutes
2. Put job script with JA token into queue
3. Start JobAgent
4. Use JA token to connect and get work
5. Get job-specific certificate (user+job ID)
6. Connect with job token, do some work

2. Data Access Tokens (Envelopes) – Interaction with Storage

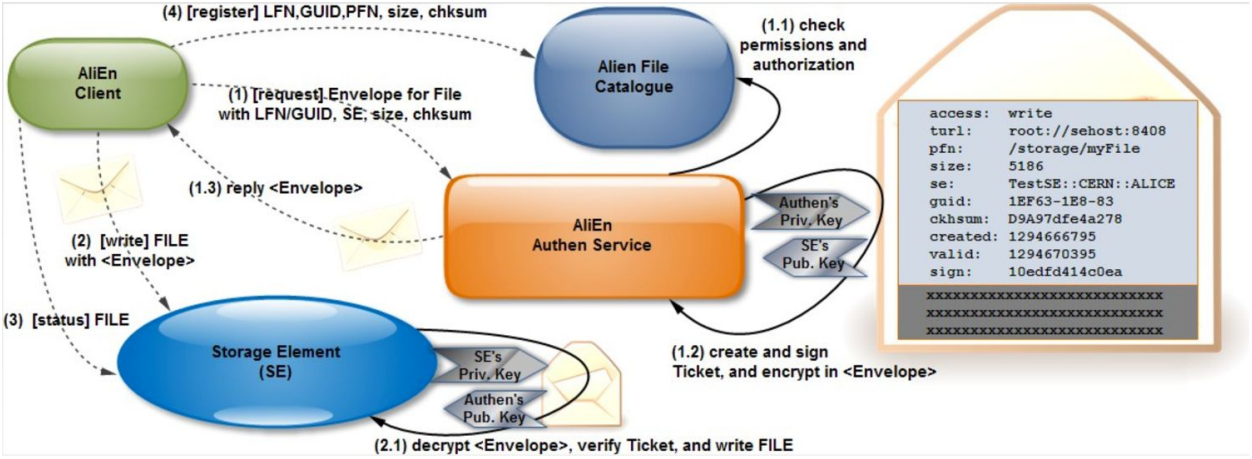
For an ALICE entity to do any kind of data related operation, it needs to have correct credentials to do such operation, and authorize against the central services, as described in point 1.

If the user has the right to perform the operation, and after checks of different types (file permissions, quotas, etc…) the central services generate a so-called envelope. An envelope is a token containing information related to the type of operation, metadata about the file, and authentication and authorization information. They are XML-encoded, and transferred to the storage via xrootd. For xrootd to understand this type of tokens, our storages have to install a small plugin that takes care of parsing the envelopes and validating them.

We have a key-pair that belongs to the central services (a public and a private key) and each storage has its own key-pair. The public key of the central services (CS) is distributed to all the storages, and the central services have the public keys of the storages. When a user requests for example a read operation, such request is validated against various criteria , and the information that will be put into the envelope is prepared: path, size, md5sum, owner, … A signature for that information is added by hashing using the SE public key. Then the whole envelope is encrypted with the CS private key.

The read request arrives at the storage and the envelope is analyzed. First it is decrypted with the public key of the CS, so only the SE instances under our control have the key and can do this. Then the content is verified with the signature, to validate this request is for the given SE by calculating the hash match using the private key of the SE.

This way we verify the request comes from an ALICE central service and is directed to the storage in question. Also we can make checks on the metadata of the file. The schema can be seen in the next diagram:



dCache macaroons (Paul)

dCache has added support for ***macaroons*** as bearer tokens that act as pure authorisation token. Perhaps the major difference between macaroons and JWT is that macaroons support autonomous delegation with attenuation. This means that any client that possesses a macaroon can create a new, derived macaroon that is more limited. In contrast, a client with a JWT must interact with the service that generated the JWT in order to acquire a new, more limited JWT token. A typical use-case where this is useful is when a service is given a macaroon and wishes some external agent to act on its behalf, but in a more limited fashion.

Macaroons contain zero or more ***caveats***. Autonomous delegation with attenuation is achieved by making it “easy” to creation a new macaroons with additional caveats, while making it “cryptographically hard” to create a new macaroon with fewer caveats. In simple terms, anyone can add a caveat, but no one can remove a caveat.

This ability to add caveats means that caveats should be thought of not as authorising activity, but rather as adding restrictions. For example the caveat:

```
activity:DOWNLOAD,LIST
```

should not be thought of as authorising downloads and directory listing, but rather denying all other activities.

In general, all caveats in a macaroon take effect. For example, a macaroon with two caveats:

```
activity:UPLOAD,DOWNLOAD,LIST
activity:DOWNLOAD,LIST
```

limits the bearer to downloading and directory listing, even though the first caveat does not impose that restriction.

If this macaroon is used to create a new, derived macaroon with the additional caveat `activity: UPLOAD,LIST` then the result is a macaroon will have three caveats:

```
activity:UPLOAD,DOWNLOAD,LIST
activity:DOWNLOAD, LIST
activity:UPLOAD,LIST
```

The bearer of this macaroon may only list directories, even though the last caveat does not impose this restriction.

All caveats supported by dCache have the form `key:value`, where `key` determines the kind of restriction being placed and `value` qualifies that restriction. Here are the different kinds of caveats that dCache supports.

The root caveat

Format `root:<path>`
Example: `root:/path/to/directory`

This is roughly equivalent to the chroot command. All client request paths are appended to `<path>` with no way of escaping; e.g., with paths like `../foo`. Multiple `root` caveats combine to form the concatenation of their paths.

The home caveat

Format: `home:<path>`
Example `home:/Users/paul`

This encodes the desired initial current directory for the user. Although this caveat is supported, as only the WebDAV door supports macaroons and this door does not react to home directories, this caveat currently has no effect.

The path caveat

Format: `path:<path>`
Example `path:/data/2017`

This limits the path elements that are visible and accessible, with all other path elements in that path being inaccessible. In contrast to the `root` caveat, the `path` caveat does not change the path of a file or directory.

For example, with the caveat `path:/data/2017`, a directory listing of `/` yields a single directory entry `data`, and a directory listing of `/data` yields a single directory entry: `2017`. Even

if there is other files and directories in `/` and `/data`, they are neither visible or accessible. There are no restrictions for directory elements beneath the `2017` directory.

The before caveat

Format: `before:<timestamp>`
Example: `before:2017-12-14T14:22:05.620Z`

This limits the validity of the macaroon. The timestamp is ISO 8601 in "Zulu" time (UTC). If there are multiple `before` caveats then the time must satisfy all caveats.

The ip caveat

Format: `ip:<ip/subnet>[,<ip/subnet>]...`
Example: `ip: 131.169.214.0/24`

This caveat limits the IP address of the bearer. Multiple IP addresses or subnets (both IPv4 and IPv6) may be specified in a single caveat: the bearer's IP address must be in at least one of the listed subnets. If the are multiple `ip` caveats then the bearer's IP address must satisfy all caveats.

The activity caveat

Format: `activity:<activity>[,<activity>]...`
Example: `activity:DOWNLOAD,LIST`

This caveat provides coarse-grain authz limitations, describing which operations the bearer is allowed to perform. Available activities are LIST (list contents of a directory), DOWNLOAD (read contents of a file), DELETE (remove a file), UPLOAD (create a new file; overwriting also requires DELETE), MANAGE (move/rename files and directories, overwriting existing content requires DELETE), READ_METADATA (discover details of file or directory), UPDATE_METADATA (modify metadata: permissions, timestamps, ACLs, …). READ_METADATA is implied if any other activity is present. If there are multiple `activity` caveats then the client's request must satisfy all caveats.

There is one additional caveat: `id`. This encodes the uid, gid of newly created files, additional gids for additional authz and a username, used for traceability. This caveat is included automatically when a user requests a macaroon, copying the information from that user. There must be exactly one `id` caveat, otherwise the macaroon is rejected.

Obtaining a macaroon

A user makes an HTTP `POST` request to dCache WebDAV interface, with `Content-Type: application/macaroon-request`. The request must be using an encrypted channel

(https://) and the request must be authenticated. Any authentication mechanism is allowed: username+password, X.509 (with or without VOMS), OpenID-Connect, Kerberos, and also macaroon.

The POST request may contain a JSON object describing the desired duration and any additional caveats -- although this is not strictly necessary (a client can add any caveats autonomously), it makes acquiring a macaroon easier and adds an extra layer of security since only the restricted macaroon is sent.

If the POST request targets a non-root path, then the macaroon will include a `path` caveat of the request path. This restricts the visible part of the namespace to this path. See `path` caveat above for details.

If the POST request is authorized via a macaroon then all caveats from that macaroon are copied to the new macaroon before adding any requested, additional caveats. This provides a mechanism by which a client can acquire a more limited macaroon without manually manipulating the macaroon itself.

Using a macaroon

The bearer can present the macaroon as a bearer token, using the `bearer` authorisation scheme in the HTTP Authorization header; e.g.,

```
GET /data/2017/my-data http/1.1
Host: dcache.example.org
Authorization: bearer MjI6MD[...]EOV6bCAo
```

For clients where this is difficult to achieve (e.g., when specifying a URL), the authz query parameter may be used; e.g.

```
GET /data/2017/my-data?authz=MjI6MD[...]EOV6bCAo http/1.1
Host: dcache.example.org
```

Both methods that a client may used to present a macaroon (within the Authorization header and the “authz” query param) are currently allowed in unencrypted channels.

EGI (Nicolas)

The EGI Check-in service aggregates information from different attribute authorities in order to create a “composite” user identity that also combines the attributes retrieved from the user’s home organisation. This “composite” user identity is then made available to EGI services enabling them to make the appropriate authorisation decisions. Specifically, the attributes that

can be used by SPs to control access to resources convey two types of information about the authenticated user, namely Entitlements and Level of Assurance (LoA).

Entitlements

User entitlements indicate a set of rights to specific resources. In the case of SAML 2.0, entitlements are expressed as `eduPersonEntitlement`[1] (ePE) attribute values, whereas in OpenID Connect via the `edu_person_entitlements`[2] claim. While ePE values can be either URLs or URNs, EGI Check-in has adopted URNs which are currently more commonly used by existing IdPs/AAs/Federations and can easily support scoping following a hierarchical structure. For this purpose, the Middleware Architecture Committee for Education (MACE) has delegated the operation of the `urn:mace:egi.eu` namespace to EGI. Using the namespace identifier registry delegation model, URN values can thus be managed in a distributed fashion by different EGI issuing authorities, communities/VOs, group management systems.

Entitlements can either refer explicitly to the protected resources in question, or implicitly by conveying the user's VO/group membership and role information (group- and/or role-based access control).

Resource-specific entitlements

A resource-specific entitlement represents the right of a user to access a particular resource. For example, the `urn:mace:egi.eu:aai.egi.eu:rcauth` value is currently being used to indicate that the holder of this entitlement is eligible for accessing the RCauth.eu Online CA service. The EGI AAI URN registry[3] lists all supported entitlement values.

Note that the resource-specific entitlements are meant to be used to grant access to specific EGI central services rather than distributed services, such as HTC or cloud resources, for which authorisation is typically based on group membership.

Entitlements expressing VO/group membership and role information

To express VO/group membership and role information for use within the EGI environment, each entitlement value represents a particular position of the user within a VO. A user may be member or hold more specific roles within the groups associated to a VO. Groups are organised in a tree structure, meaning that a group may have subgroups, which in turn may have subgroups, etc. This hierarchical structure implies that if someone is member of a subgroup, then they are also member of the parent group.

---

[1] http://software.internet2.edu/eduperson/internet2-mace-dir-eduperson-201602.html#eduPersonEntitlement

[2] There is currently no standard OpenID Connect claim to express the `eduPersonEntitlement` attribute. However, the REFEDS OpenID Connect for Research and Education Working Group (OIDCre) is already investigating the standardisation of new claims for expressing the attributes defined in the eduPerson schema, including the `eduPersonEntitlement`.

[3] https://wiki.egi.eu/wiki/URN_Registry:aai.egi.eu

Specifically, the `eduPersonEntitlement` values expressing VO/group membership and role information adopt the following formatting specification:

`urn:mace:egi.eu:<authority>:[<group>[:<subgroup>:…]]:<role>@<vo>`

where:

- `<authority>` identifies the authoritative source for the entitlement value
- `<vo>` is the name of the Virtual Organisation
- `<group>` is the name of a group in the identified VO; specifying a group is optional
- zero or more `<subgroup>` components represent the hierarchy of subgroups in the `<group>`; specifying sub-groups is optional
- the `<role>` component is scoped to the rightmost (sub)group; if no group information is specified, the role applies to the VO

Note that the above syntax will be adapted to follow the AARC "Guidelines on expressing group membership and role information (201710)"[4].

Level of Assurance

Based on the authentication method selected by the user, EGI Check-in assigns a Level of Assurance (LoA), which is conveyed to the SP through either the `eduPersonAssurance` attribute and the Authentication Context Class (`AuthnContextClassRef`) of the SAML authentication response, or using the `acr` claim in the case of OIDC services. While the EGI AAI currently distinguishes between three LoA levels, namely `Low`, `Substantial` and `High`, it is planned to support the REFEDS Assurance Framework (RAF)[5], which allows for both a composite assurance level/profile and for assurance component values to be expressed. In the RAF, it is the component values that play the principle role in expressing assurance information, and the composite profiles (e.g. "Cappuccino" and "Espresso") are the result of a specific combination of assurance components.

INDIGO-Datacloud AAI and Identity and Access Management (Andrea)

The AAI developed in INDIGO DataCloud is based on: OAuth, for authorization, and OpenID Connect for exposing authentication information to services.
The INDIGO Identity and Access Management (IAM) service is responsible for user registration, authentication, and for providing the abstraction of VO/Collaboration to relying services.
Relying services are registered in IAM as OAuth/OpenID connect clients, and use standard OAuth/OpenID connect flows to obtain access to user authentication/authorization information.

[4] https://aarc-project.eu/wp-content/uploads/2017/11/AARC-JRA1.4A-201710.pdf
[5] https://wiki.refeds.org/display/GROUPS/Assurance+Working+Group

Authentication information is exposed to services via OpenID connect standard interfaces (signed JWT Id tokens and /userinfo endpoints). Authorization information is exposed to services via signed JWT access tokens.

JWT access tokens, in the default configuration, contain the following claims:

- sub: an opaque, persistent unique identifier for the user defined at the IAM VO level (in case of a user token); the OAuth client id (in case of a client token, i.e. a token issued to a client application, not bound to a specific user)
- iss: this is the token issuer
- aud: the audience for the token (i.e. the services meant to accept it)
- exp: token expiration time
- iat: token issued at time

More detailed authn/authz information can be obtained via standard OpenID Connect /userinfo and OAuth token introspection endpoint, like, for instance:

- groups: group information
- email: user email
- organization_name: this is the VO/collaboration name
- scope: scopes bound to the token

IAM provides a registration service that can be used to manage user collaboration enrollment, following a registration flow very similar to the one used in production by WLCG VOs (and implemented in VOMS Admin, which was developed by the same team behind IAM).
IAM also provides an enrollment flow that supports the automatic creation of user accounts when the user authenticates with a trusted IdP. This mechanism could be used, for instance, to directly integrate with the CERN SSO and generate automatically users for authenticated users that come with the right attributes (e.g., the user is registered in CERN HR db as a member of the ATLAS experiment).
IAM provides SCIM provisioning APIs that can be used to provision information about the user and the VO/Collaboration structure to relying services. These APIs could be used, for instance, to expose in standard, RESTful manner information about users to experiment frameworks (like ATLAS AMI).
IAM integrates nicely with the INDIGO WaTTS (the Token Translation Service) which provides the ability to translate OAuth/OIDC tokens and related information to other types of credentials (e.g., X.509 certificates, S3 keys, etc.).

IAM provides a scope policy API that can be used to restrict which users have access to specific scopes, in order to have central control on how scope-based authorization can be implemented at relying services. This API allows, for instance, to restrict which VO/collaboration users/groups will be entitled to request the scope “write-files” and “submit-job” and have such capabilities linked to an access token issued by the IAM.

SciTokens (Brian)

(Note: this section used to be called “OSG”, but these are separate projects)

The SciTokens JWT profile provides a description of a capability-based token that allows for distributed verification. The aim is to allow entities to describe certain authorizations for the bearer in a way that they can be implemented by remote grid resources.

The profile is outlined in this document: https://scitokens.org/technical_docs/Claims (and summarized here)

The claims used are relatively standard:

- "sub". Required for SciTokens. Treated as an opaque string managed by the issuer, with the restriction that a unique subject name may not map to more than one identity. For privacy reasons, a single identity may result in one or more "sub".
- "nbf" / "exp". Required for SciTokens, but the meaning is unchanged from the JWT RFC.
- "iss". Issuer endpoint. Must support endpoint auto-discovery as outlined by the OpenID Connect standard (or the upcoming OAuth auto-discovery).
- "jti". Meaning unchanged from the RFC. Use is encouraged for auditing and tracing, but optional.
- "aud". Restriction on the resources that the token is allowed to access. What "aud" a service self-identifies as is left up to the issuer. If the site T2_US_Nebraska has an endpoint at https://red-gridftp.unl.edu, the endpoint may decide to accept tokens with "aud" set to either "T2_US_Nebraska" or "https://red-gridftp.unl.edu".
- "scp" (scopes). The authorizations that the bearer has; the claim name is taken from the draft RFC on token exchange. The scopes in use for SciTokens are a bit non-standard to reflect the fact we may have filesystem-like resources. More below.

Token verification adheres fairly close to the standards. A few notes on deviations:

- We require the use of auto-discovery (OIDC defines this, as well as a draft for OAuth2 https://tools.ietf.org/html/draft-ietf-oauth-discovery-07) to establish a chain of trusts. Hence, the trust root for the ecosystem is the standard TLS trust roots (CAs), which allows the service to discover and download the signing public keys from the issuer.
    - So, each token can be verified and validated based on.
- We currently require all claims to be understood by any entity validating a token: this is fairly non-standard, but as any additional claim in an authorization token tends to restrict authorizations, we feel it would be a mistake to ignore them. Worth revisiting.

The heart of the SciToken approach that is different from any standards is the naming of the scopes. For this, we split our scopes into two parts: authorization and resource paths. Authorizations are along the lines of:

- Read
- Write
- Queue
- Execute

Whereas resource paths provide some additional limitation. For example the scope “read:/” authorizes the bearer to read any file inside the issuer’s resource area; “read:/foo/bar” restricts the authorization to any file or directory under “/foo/bar”.

We’ve gone a bit back and forth on whether to use URN-style scopes (https://scitokens.org/v1/read) or just direct ones (read) and currently are leaning toward the shorter variant, staying with the tendency of JWT to shorter strings.

Given that most of the claims are standardized, I see the challenges in creating a WLCG profile in:

- Separating out the use case of an “identity token” versus “authorization token”.
- Defining the scopes we want and their approximate meaning. For example, SciTokens just has a “write” scope that probably needs to be broken into smaller pieces.
    - For reference on why this is difficult, look at capabilities in the Linux kernel. :/
- Resolving some small details on how to best do validation.

There are some questions about what workflows (other than standard OAuth2!) we may want to generate tokens - but seems out-of-scope for now.

Input from the AuthZ Interop WG (XACML attributes) (Mischa/Brian)

Comparison Table

Infrastructure or Project	**Token Format**	**Token Contents**	**Token validation**	**AuthZ is resource specific (i.e. knows where it’s going)**	**AuthZ is user oriented (i.e. conveys user’s roles and groups)**	**Delegation permitted**	**Comment**

ALICE Token Certificate	X509	User ID, token resubmission #, …	Existing grid infrastructure (CAs)	Yes	No	No	Production early 2018
ALICE Data Access Token	Encrypted XML	Authorisation, validity, Token ID(?)	PKI	Yes	No	No	
dCache	Macaroons (similar to JWT)	Caveats, including ID caveat	Requested over encrypted channel Verification by client (calculating a hash value, over all caveats, starting with a secret value known by issuer and client.)	Yes	No	Yes, supported by default	Functionality seems restricted to file read/write/list operations
EGI Checkin	JWT over OIDC	Entitlements, LoA (User ID? R&S bundle?)	Verification with issuer when used (?)	Yes	Yes	~~Yes~~No Not implemented yet	
EGI Checkin	SAML	Entitlements, LoA (User ID? R&S bundle?)	Verification when issued	Yes	Yes	No(?)	

Indigo IAM	JWT over OAuth/OIDC flows	Base token = opaque ID, issuer, audience, expiration. Extra information queriable = groups, email, VO, scopes, LoA	IAM supports local token verification (based on JWT validation) and/or remote token verification (via userinfo and token introspection endpoint)	Yes	Yes	Yes, return to token issuer	
SciTokens	JWT	Opaque ID, issuer, audience, some JWT things, scopes	Verification with issuer when used (?)	Yes	No	?	Services understand all jwt scopes, in case there is an additional restriction
XACML							
Legacy WLCG	*VOMS proxy*	*Groups, role, DN, issuer, validity*	*Existing grid infrastructure (CAs)*	*No*	*Yes*	*Yes*	

Challenges

- Defining scopes
- Separating authentication and authorisation claims
- Validation

References

https://wiki.refeds.org/display/GROUPS/OIDCre REFEDS OIDC Working Group

https://tools.ietf.org/html/rfc7662 OAuth 2.0 Token Introspection