



Rapporti Tecnici INAF INAF Technical Reports

| | |
|------------------------------------|--|
| Number | 37 |
| Publication Year | 2020 |
| Acceptance in OA@INAF | 2020-09-10T13:18:56Z |
| Title | Rebuilding services A&A architecture using OAuth2, OIDC and JWT |
| Authors | ZORBA, SONIA; TINARELLI, FRANCO; KNAPIC, Cristina; Butora, Robert; TAFFONI, Giuliano; MAJOR, BRIAN |
| Affiliation of first author | O.A. Trieste |
| Handle | http://hdl.handle.net/20.500.12386/27284 ; http://dx.doi.org/10.20371/INAF/TechRep/37 |



Rebuilding services A&A architecture
using OAuth2, OIDC and JWT

| | |
|----------------|-------------|
| Issue/Rev. No. | 1.0 |
| Date | Sep 2, 2020 |
| Page | 1 of 16 |


Rebuilding services A&A architecture using OAuth2, OIDC and JWT

Issue/Rev. No.: 1.0

Date: September 2, 2020

Authors: Sonia Zorba, Franco Tinarelli, Cristina Knapic, Robert Butora, Giuliano Taffoni, Brian Major

Approved by: Sara Bertocco, Marco Molinaro

| | | | | | | | | |
|---|--|---|----------------|-----|------|-------------|------|---------|
|  | Rebuilding services A&A architecture using OAuth2, OIDC and JWT | <table border="1"> <tr> <td>Issue/Rev. No.</td> <td>1.0</td> </tr> <tr> <td>Date</td> <td>Sep 2, 2020</td> </tr> <tr> <td>Page</td> <td>2 of 16</td> </tr> </table> | Issue/Rev. No. | 1.0 | Date | Sep 2, 2020 | Page | 2 of 16 |
| Issue/Rev. No. | 1.0 | | | | | | | |
| Date | Sep 2, 2020 | | | | | | | |
| Page | 2 of 16 | | | | | | | |

Contents

| | | |
|----------|--|-----------|
| 1 | Understanding A&A | 3 |
| 1.1 | Introduction | 3 |
| 1.2 | Externalizing authentication | 3 |
| 1.3 | IDEM and eduGAIN | 3 |
| 1.4 | Account linking and merging | 4 |
| 1.5 | OAuth2 and OIDC | 5 |
| 1.6 | Token types | 5 |
| 1.7 | JSON Web Tokens (JWT) | 7 |
| 1.8 | JSON Web Key Set (JWKS), key distribution and key rotation | 8 |
| 1.9 | Refresh tokens | 8 |
| 1.10 | Credential delegation | 9 |
| 2 | IA2 use cases and implementations | 10 |
| 2.1 | RAP | 10 |
| 2.2 | Grouper and GMS | 11 |
| 2.2.1 | Membership transfer for account merging | 11 |
| 2.3 | Portals, UserSpace and FileServer | 12 |
| 2.4 | Non-browser access | 13 |
| 2.5 | Apache Guacamole | 13 |
| 2.6 | VLKB | 14 |
| 3 | Conclusions | 14 |
| 3.1 | Lessons learned | 14 |
| 3.1.1 | SAML non-browser access is even worse than OAuth2 one | 14 |
| 3.1.2 | Account merging is not trivial | 15 |
| 3.1.3 | Think twice before changing A&A | 15 |
| 3.2 | Further developments | 15 |



1 Understanding A&A

1.1 Introduction

In the last years the Italian Center for Astronomical Archives (IA2) team performed a lot of changes in its authentication and authorization infrastructure, trying to implement an approach valid for all the services it manages. Over time several challenges emerged, bringing to reconsider some initial choices, as requirements became clearer.

This document summarizes the status of the current A&A architecture, considering all the use cases handled by IA2, but also highlights the pros and cons of the discarded approaches.

Since A&A is a quite complex topic the following subsections provide an introduction to some concepts needed for understanding the architectural choices we made.

1.2 Externalizing authentication

Authentication is the process used to understand who the user is (identity verification). Perhaps the most used approach consists in providing a username and a password to a user. These credentials can be created and managed by the same organization providing the services the user would like to use. However in the last years more and more providers started adopting externalized authentication protocols, like SAML (Security Assertion Markup Language) and OIDC (OpenID Connect). The advantages of this are the possibility for the users to use the same credentials to login on different services (Single Sign-On) and the minimization of password exposure (password is stored only by the identity provider and it is never shared to third-party applications). The main drawback of externalization is an increased complexity in handling the authentication outside the browser (both SAML and OIDC are designed mainly for being used inside a browser) and this can be critical for the astronomical community, since a lot of web services are accessed in a programmatic way.

1.3 IDEM and eduGAIN

IDEM¹ is the Italian national federation for universities and research institutions for authentication and authorization. This federation deals with registration of Identity Providers (IdP) and Service Providers (SP) belonging to Italian research institutions. These providers communicates together using SAML.

An Identity Provider, as the name suggests, is a service able to provide user authentication (in practice, it displays the web form in which the user inserts him/her username and password). A Service Provider is a service able to initiate a SAML flow in order to allow an user to perform a login and then use the service. This flow usually includes a Discovery Service, which provides a dropdown menu for selecting an IdP.

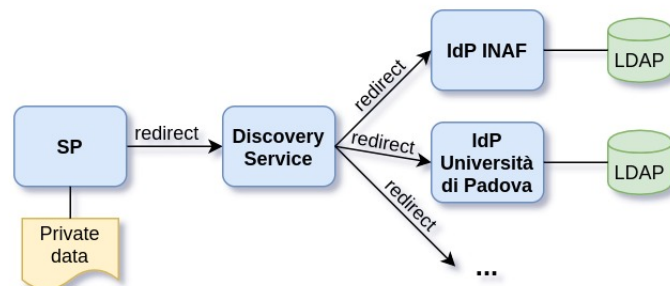


Figure 1: Typical interaction between SAML entities. The Service Provider needs to verify the user identity before providing access to the private data. Identity Providers are usually connected to a LDAP, a hierarchical database containing user password and profile information.

¹<https://www.idem.garr.it/en/>

EduGAIN² is an international interfederation service interconnecting research and education identity federations, or, in other words, a federation of federations. IDEM is part of eduGAIN. All INAF users can login with their institutional credentials using INAF IdP.

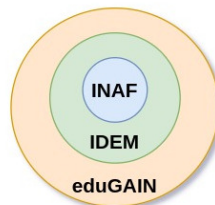


Figure 2: EduGAIN is an international interfederation, IDEM is a national federation of research institutions, INAF IdP is part of IDEM.

1.4 Account linking and merging

According to AARC³ guidelines “identity linking (also known as account linking) refers to the process of connecting the user’s infrastructure identity with their external identities, i.e. identities created and assigned by Identity Providers that reside outside of the administrative boundaries of the infrastructure, such as institutional IdPs or social media IdPs[1].”

This definition assumes that users always own a local account and, as an additional feature, they can associate it to one or more external accounts. If the local account is created “on the fly” (allowing users to register into the system directly using an external provider) and several external providers are accepted, users could perform different registrations with multiple credentials. This can be an advantage for some users, especially for scientists belonging to multiple research institutions or for those who would like to exploit the benefits of social logins (usually based on long lasting cookies, which make the login feasible with just a couple of clicks). However this leads to a proliferation of local accounts, that can be solved with a procedure we could call “account merging” (or joining).

Difference between account linking and account merging can be subtle, so a more formal definition is provided (see also Figure 3):

- **account linking:** the act of associating one local account to one or more external accounts (locally we have always one internal user identifier)
- **account merging:** the act of merging two local accounts (each one possibly associated to one or more external accounts) into a single local account (locally we start with two internal user identifiers and we end up with one)

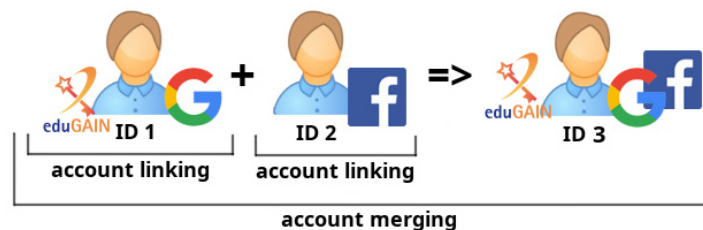


Figure 3: Account linking vs account merging. Local account ID 1 is linked with a Google account and an eduGAIN account; local account ID 2 is linked with a Facebook account; ID 1 and ID 2 are merged into a new local account ID 3 that is linked with all the external accounts.

²<https://edugain.org>

³<https://aarc-project.eu>



1.5 OAuth2 and OIDC

OAuth2 (RFC 6749)[5] is an authorization protocol. Several flows are possible but when a user is involved the general process is based on a sequence of redirects (see Figure 4). The user (also called resource owner) goes to the client application, then there is a redirect to an authorization server where the user inserts the credentials and after that the user is redirected back to the client application. During the last redirect the client application receives an access token that it can use to access a protected resource owned by the user. It is also possible to receive a refresh token, that can be used to request a new access token when the previous one expires.

Since the second step consists in the insertion of credentials it may be tempting to use an access token (that is built for authorization) for authentication purposes. This technique is called “Pseudo-Authentication” and it is discouraged, since it could lead to provide an application more privileges than it needs. Suppose for example to have an access token that grants access to a set of private files. This token may also contain authentication information (like the subject id), so it is possible to reuse the same token for retrieving only that kind of information, ignoring the fact that the token also provides access to private files. Malicious applications could exploit the fact that someone provided them tokens containing more privileges than they need. So, the correct way to handle authentication with OAuth2 is providing an access token that is specifically built for authorizing only the retrieval of user profile information from a specific “userinfo” endpoint. Notice that OAuth2 standard defines nothing about this endpoint, so it is application-specific or implies the usage of proprietary API. For this reason OpenID Connect standard was created[2]. OIDC provides an identity layer on the top of OAuth2, so it standardizes the authentication as an extension to the OAuth2 authorization process. It defines a specific token, called “ID Token”, that is provided together with the access token and contains information about the identity of the user. Using two different tokens ensures that authentication and authorization are kept separated. As access tokens should not be used for checking authentication, ID tokens should not be used for authorization[3]. OIDC specify that ID tokens must use JWT (JSON Web Token) format (see subsection 1.7).

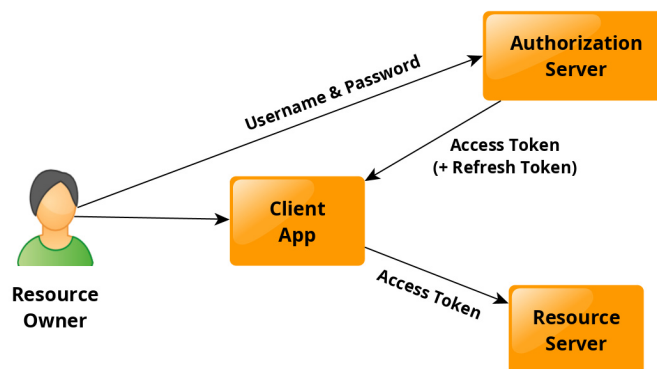


Figure 4: A simplified view of the OAuth2 protocol

1.6 Token types

OAuth2 doesn't set constraints about token format, however most of the tokens are “bearer tokens” (RFC 6750)[6]. Bearer tokens can be used by any party to get access to the associated resources without demonstrating possession of a cryptographic key. So, to prevent misuse, bearer tokens need to be protected from disclosure in storage and in transport (this usually means using HTTPS protocol).

When a resource server receives a bearer token it usually doesn't know which client sent it, unless additional client authentication mechanisms have been set up. Indeed, any client can send a bearer token simply adding it to the HTTP Authorization header, without knowing any additional information:

```
Authorization: Bearer <token>
```



Notice that OAuth2 specification describes client authentication, but that authentication is between the client and the authorization server, not between the client and the resource server. Moreover, in many cases OAuth2 client authentication is not reliable. OAuth2 distinguishes between “confidential clients” and “public clients”. Public clients are clients incapable of maintaining the confidentiality of their credentials, like native applications (desktop or mobile applications) and JavaScript applications.

An alternative to setting up some authentication mechanisms between the client and the resource server is using MAC tokens instead of bearer tokens. MAC tokens are specifically designed to require clients demonstrating the possession of a cryptographic key[16]. MAC tokens are not widely used and the specification describing them is still a draft.

MAC stands for Message Authentication Code and indicates hash functions that take as input both a message and a secret key. In case of OAuth2 MAC tokens, the secret key is a temporary key (called also session key) provided by the authorization server to the client together with the access token. The client adds the computed hash to the requests it makes to the resource server, in order to demonstrate the possession of the session key.

Moreover, we can classify the tokens in 2 categories:

- reference tokens: tokens which are just an handle, a random string, and it is necessary to do an additional call to the authorization server in order to validate them and retrieve their content (the authorization server provides a “token inspection” endpoint);
- self-contained tokens: tokens which contains all the information inside and are signed or encrypted, so they can be validated locally by the receiver. JWT are an example of self-contained tokens (see next section for additional details about this kind of tokens).

Notice that keys used for signing or encrypting self-contained tokens are not the same keys used by MAC tokens. Indeed MAC tokens are used to verify the client identity, while self-contained tokens usually only guarantee that they have been issued by a specific authorization server (clients can still be anonymous from resource server perspective).

Moreover, in most of the cases, clients don’t need neither to validate the tokens they receive nor to understand their content: they only have to use them to retrieve resources from the resource server. So, usually, the validation/decryption of self-contained access tokens is done by the resource server, not by the client. In case of OIDC, the ID token content is instead extracted by the client (in fact you don’t have to access “resources” with it).

While bearer tokens can be both reference tokens or self-contained tokens, MAC tokens are always JWT. MAC tokens are encrypted using a key that is shared between the authorization server and the resource server, so the client is not able to decrypt them. The encrypted payload of a MAC token contains also the session key. In this way the resource server is able to validate the hash sent by the client.

Here there is an example of what a client receives from the authorization server when using MAC tokens:

```
{
  "token_type": "mac",
  "expires_in": 3600,
  "mac_key": "<random string used to compute the hash (MAC)>",
  "mac_algorithm": "hmac-sha-256",
  "access_token": "<JWT that can be decrypted by the resource server;
                  it contains also the mac_key>",
  ...
}
```

1.7 JSON Web Tokens (JWT)

JSON Web Tokens (RFC 7519)[7] are strings composed by 3 parts: header, payload and signature. Each part is encoded in URL-safe Base64 and separated by a period ('.') character; this leads to a compact representation. Both header and payload are a JSON structure, but payload can also be encrypted. Signature and encryption can use different algorithms; chosen algorithm is specified in the JWT header.

Encoded PASTE A TOKEN HERE

```

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6ImU2NzAzNWw4NmFkMmExMzgifQ.eyJpc3MiOiJzZC28uaWEyLmluYWYuaXQiLCJzZW50IiIyMzg2IiwiaWF0IjoxNTg0ODQ0NzY3LCJleHAiOjE1ODg5MzExNjcsImF1ZCI6ImdtdcyJ9.jotiy3p0brpmheA4CQ0IBSagsQH0e_d0s1sgygX9HEvQtha2yn0zBQ9PpRW51p48BCPAopdf0lee6fQ20s-JiehfftgnJ755-b634TezXAbd7dNwPdE7yWP--Qia73hcbpnZLY1YPFXI8LUyUoqeLtBEgF7Kkb35v3i0qdoFSOP1ry3jMUR5ofIgwS901_DI8GcEAHc9VayPtE9vHc22gne9ZDYvkYS91iUv7QyRDSEBXubprjxh3fezf5GmNa1UWQdUqEqHp4du0tT4CefoANci4TFqAsGaqxmcyaK2poc-kfE6UY4U6ofynW1oh1WVB36vpvb07ZMApZ8jcr8Ng
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "RS256",
  "kid": "e67035c86ad2a138"
}
```

PAYLOAD: DATA

```
{
  "iss": "sso.ia2.inaf.it",
  "sub": "2386",
  "iat": 158844767,
  "exp": 1588931167,
  "aud": "gms"
}
```

VERIFY SIGNATURE

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
```

Figure 5: Online JWT decoder at <https://jwt.io/> displaying the content of a JWT

Keys of the JSON object composing the payload are known as “claims”. Some claims are standardized and commonly used, like:

- “iss” (issuer): who created and signed the token;
- “sub” (subject): whom the token refers to (usually the user id);
- “aud” (audience): who or what the token is intended for (usually the identifier of the service which will receive requests having that token in the header);
- “iat” (issued at): token creation time - Unix timestamp;
- “exp” (expiration time): Unix timestamp;
- “jti” (JWT ID): unique identifier for the JWT (used to avoid token reuse).

Moreover, it is possible to add custom claims in the payload.



1.8 JSON Web Key Set (JWKS), key distribution and key rotation

In JWT, signatures can be generated both using symmetric keys (e.g. HMAC) and asymmetric ones (e.g. RSA). Services must know these keys in order to be able to verify token signatures. For distributing keys across multiple services, token issuers can provide a JSON Web Key Set (JWKS) endpoint, as defined in RFC 7517[9].

The JWKS endpoint returns a JSON structure containing all the keys used to generate signatures. If symmetric keys are used they must be encrypted in order to prevent disclosure, instead, if asymmetric keys are used, it is possible to expose to everybody the public keys.

It is possible to have multiple coexisting valid keys, distinguished by a key id (kid). In order to select the proper key for performing signature validation, services can check the kid claim contained in the JWT header, since it identifies the key that has been used to sign the token.

Listing 1: Example of a JWKS endpoint response containing one public key. The “kty” (key type) parameter states RSA algorithm is used. The “use” parameter set to “sig” indicates the key is used for signing tokens. Alternatively, the parameter can be set to “enc” to indicate that the key is used for encrypting tokens. Parameters “n” and “e” represent modulus and public exponent of the RSA key encoded in Base64.

```
{
  "keys": [
    {
      "kty": "RSA",
      "kid": "e67035c86ad2a138",
      "use": "sig",
      "n": "6R-qAvAPQNrx [...] VAfWqevpJc0RezVsQ==",
      "e": "AQAB"
    }
  ]
}
```

To increase security, keys should be changed periodically. However, if a key is suddenly removed from a JWKS and the authorization server has just issued tokens signed with that key, services will stop accepting those tokens, even if they are not expired yet. The appropriate way to handle key updates is performing a procedure called “key rotation”. It consists in keeping always at least 2 keys in the key set: when you have to update a key, a new key is generated and all the new tokens are signed with that key, but the old one is not removed from the key set until all the tokens previously signed with it are expired.

1.9 Refresh tokens

Self-contained tokens like JWT are very good to achieve scalability, since they can be validated without performing additional calls. The main issue arising with this approach is that these tokens can't be revoked (this could be very important if a malicious entity steals the token). For this reason tokens have usually a short lifespan (often one hour). However, many applications need to be able to retrieve protected resources for longer periods of time. To overcome this limitation refresh tokens can be issued.

Refresh tokens are usually reference tokens which can be invalidated by the authorization server. When a client application retrieves an access token, the authorization server can provide also a refresh token in the same response. The refresh token can be used by the client application to obtain a new access token when the old one expires. Since a call to the authorization server must be performed in order to obtain a new token, it is possible to deny the access to a compromised service in this phase.

When a new access token is obtained using the refresh token, the authorization server can provide also a new refresh token. This technique is called “refresh token rotation” and ensures that also refresh tokens have a not too long lifespan.

1.10 Credential delegation

Consider a situation where a service A needs to perform requests to another service B. Both services have restricted access, however the user interacts directly only with service A. Service B needs to know who is the user that is interacting with service A. If the two services trust each other, service A can tell to service B who the user is. However, establishing trust relationships between services leads to a non scalable and non interoperable architecture. A better approach consists in setting up a mechanism for forwarding the user authentication from service A to service B (see Figure 6). In this way services don't need to trust each other, because you can assume that it is always the user making the call. Methods used to authenticate service A on service B as if it were the user are known as “credential delegation”.

Credential delegation can be implemented using X.509 certificates, in particular generating proxy certificates signed by the client and used for service-to-service communications. This technique has been adopted by the IVOA Credential Delegation Protocol version 1.0 (IVOA Recommendation, CDP-1.0[4]) and it is described in detail inside the specification's document.

Credential delegation can be implemented also using tokens but different approaches are possible, depending on the used infrastructure and security constraints.

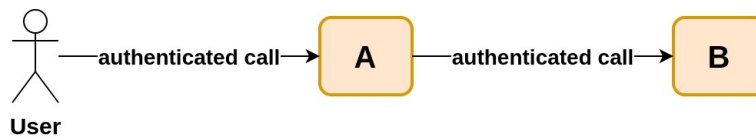


Figure 6: How to propagate authentication between services?

The easiest approach is called “token relay” and consists in simply forwarding the token from service A to service B. This is not very secure, indeed it is quite similar to forwarding a password from one service to another, with the difference that a token usually has a short expiration time.

Some systems deny this possibility because each service is configured to accept tokens explicitly generated for it, for example validating the “audience” (aud) JWT claim. Moreover, in some other systems each token is valid only for one HTTP call, because token reuse is prevented, for example associating each token with a unique identifier using the JWT ID (jti) claim.

In these cases credential delegation can be implemented using the OAuth 2.0 Token Exchange protocol (RFC 8693)[8]. This proposed standard defines an entity called Security Token Service (STS), able to exchange a token generated for a service for a new token having a different audience, scope and/or expiration time.

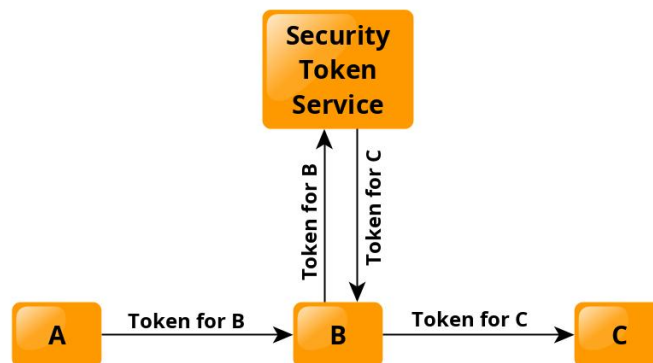


Figure 7: Service B asks to the Security Token Service a token for calling service C

OAuth 2.0 Token Exchange protocol identifies some specific claims to use for expressing delegation, like the actor (act) claim, which identifies the party acting on behalf of the subject. This claim is a JSON object in

turn and a chain of delegation can be expressed by nesting one “act” claim within another. Other claims that can be used in this context are the “client_id” claim, which is set as the client identifier of the OAuth 2.0 client that requested the token, and the “may_act” claim, which identifies the party authorized to act on behalf of another.

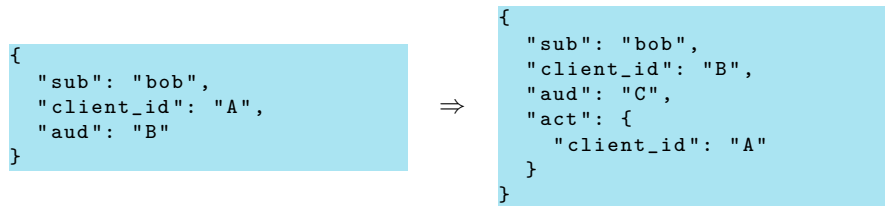


Figure 8: Usage of the actor claim in token exchange (service B exchanges token received from service A for a new token valid for service C)

2 IA2 use cases and implementations

2.1 RAP

In 2017 IA2 started adopting an application called RAP (Remote Authentication Protocol) to login users on their services. RAP was developed by Franco Tinarelli as a prototype for the authentication in the SKA (Square Kilometer Array) project and some changes were made over time in order to fit IA2 needs[10]. RAP supports different authentication mechanisms (eduGAIN, some social logins and X.509 certificates) and provides account linking and merging. At the beginning RAP sent identity information to the caller application using a custom format, then it has been standardized using OAuth2/OIDC.

Currently RAP acts as an OAuth2 authorization server, so it can provide both ID tokens and access tokens for granting access to other services. RAP will issue an OIDC token if the user reaches the RAP page after being redirected by an application in order to perform a login. Client applications can also call the RAP web service to obtain access tokens for authorization purposes and users can download similar tokens for programmatic access using the RAP Token Issuer page (see subsection 2.4).

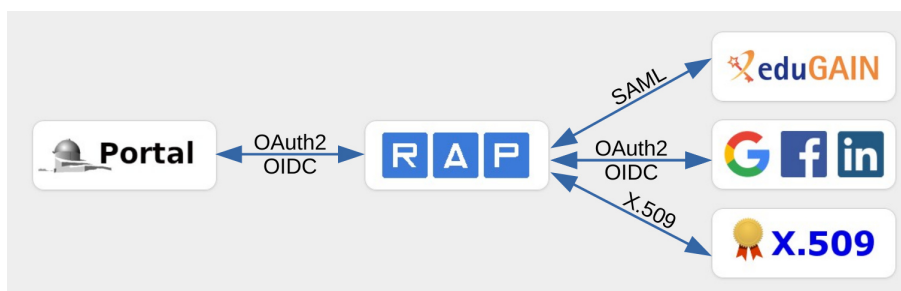



Figure 9: An application (portal) using RAP for authentication

OAuth2 defines different flows for retrieving access tokens[11]. Most of the IA2 services use the Authorization Code Grant Flow[5]. This flow implies an intermediate step in which the caller application receives a “code” (a random string) and this code is used for retrieving the access token. The code is exposed in the browser because it is sent as a request parameter, while the access token is retrieved from the code server-side only and then it is stored in the HTTP session. In this way only the server knows the access token, providing a better security. The server can also obtain new access tokens using the refresh token, if needed.

| | | | | | | | | |
|---|--|---|----------------|-----|------|-------------|------|----------|
|  | Rebuilding services A&A architecture using OAuth2, OIDC and JWT | <table border="1" style="width: 100%;"> <tr> <td style="width: 50%;">Issue/Rev. No.</td> <td>1.0</td> </tr> <tr> <td>Date</td> <td>Sep 2, 2020</td> </tr> <tr> <td>Page</td> <td>11 of 16</td> </tr> </table> | Issue/Rev. No. | 1.0 | Date | Sep 2, 2020 | Page | 11 of 16 |
| Issue/Rev. No. | 1.0 | | | | | | | |
| Date | Sep 2, 2020 | | | | | | | |
| Page | 11 of 16 | | | | | | | |

Since in many cases multiple services need to communicate together we need also credential delegation. We considered the possibility to implement the OAuth2 Token Exchange protocol, because having different tokens for each service increases the security level of the system. However, we are aware that implementing the Security Token Service and performing additional calls for exchanging the tokens implies also an increased complexity, so we concluded that in most of the IA2 use cases token relay mechanism is an acceptable trade off between security and implementation simplicity.

This doesn't mean that services are accepting all the valid tokens issued by RAP, indeed we are still validating the audience claim, however the claim value is a list of service identifiers. This is compliant with the audience claim definition provided by RFC 7519:

The “aud” (audience) claim identifies the recipients that the JWT is intended for. [...] In the general case, the “aud” value is an array of case-sensitive strings, each containing a StringOrURI value.

Moreover, the “scope” claim is used to limit token usage to a specific subset of actions.

Listing 2: Token valid for reading data both from GMS and file service

```
{
  "iss": "sso.ia2.inaf.it",
  "sub": "RAP:2386",
  "iat": 1588154088,
  "exp": 1588240488,
  "aud": ["gms","file"],
  "scope": "read:gms read:file"
}
```

2.2 Grouper and GMS

Grouper⁴ is an enterprise access management system which has been used by IA2 for managing groups which are used to define policies for controlling access to protected resources. Grouper was modified in order to work with earlier RAP versions, but some issues emerged over time, in particular regarding permission inheritance and membership transfer for account merging[12].

Since additional modifications were necessary for working with OIDC, it has been decided to implement a similar tool, following the IVOA GMS (Group Membership Service) specification, currently a Working Draft at version 1.0[13]. Unlike the current specification, authentication on GMS service is not based on X.509 certificates (as expected by IVOA Credential Delegation Protocol version 1.0[4]), but uses JWT. This possibility will probably be added in the next major revision of the CDP specification.


2.2.1 Membership transfer for account merging

RAP account merging process results in the deletion of one of the accounts, transferring all the identities belonging to it to the other one. Two accounts that are going to be merged could have separately been associated to different groups. In this case it is necessary to transfer also all the memberships from one account to the other. So, before performing an account merging, RAP needs to call the GMS for performing the memberships transfer. It follows that, for successfully implementing account merging feature, authentication and authorization services have to be somehow coupled to work together.

The membership transfer call on the GMS is performed by RAP using a JWT with an additional claim: the “alt_sub” (alternative subject). Using this claim together with the mandatory claim “sub” (subject), it is possible to identify two different accounts inside the same token, so that the GMS will be able to transfer memberships and permissions from one account to the other.

Additional considerations about account merging are presented in subsection 3.1.2.

⁴<https://www.internet2.edu/products-services/trust-identity/grouper/>

| | | | | | | | | |
|---|--|---|----------------|-----|------|-------------|------|----------|
|  | Rebuilding services A&A architecture using OAuth2, OIDC and JWT | <table border="1" style="width: 100%;"> <tr> <td style="width: 50%;">Issue/Rev. No.</td> <td style="width: 50%;">1.0</td> </tr> <tr> <td>Date</td> <td>Sep 2, 2020</td> </tr> <tr> <td>Page</td> <td>12 of 16</td> </tr> </table> | Issue/Rev. No. | 1.0 | Date | Sep 2, 2020 | Page | 12 of 16 |
| Issue/Rev. No. | 1.0 | | | | | | | |
| Date | Sep 2, 2020 | | | | | | | |
| Page | 12 of 16 | | | | | | | |

2.3 Portals, UserSpace and FileServer

Astronomical archive portals hosted by IA2 need to retrieve user groups from GMS in order to provide access to private files. Each portal interacts with other 2 services: UserSpace, used to store temporary files generated by users, and FileServer, which provides access both to public and private files.

Portals use RAP for login, so they could reuse the same access token received from RAP to perform calls on UserSpace and FileServer. However we have 3 peculiar use cases which led us to decide to give portals the ability to issue tokens in turn. So, each portal has its own JWKS endpoint and generates new tokens for the other services, often using the information retrieved from the RAP and the GMS. The FileServer retrieves keys both from portals and RAP JWKS endpoints.

The reason behind this architectural choice are listed in detail in the next paragraphs.

Anonymous users support Even anonymous users are allowed to generate temporary files from the portals and store them into the UserSpace. Portal remembers anonymous users generating a random string and storing it on a cookie. This string is then used as user id when calling the UserSpace.

Portal creates JWT for the UserSpace inserting this random string in the “sub” claim. In this way the same mechanism is used for identifying both anonymous and registered users on the services.

Local login Astronomical observatories are usually located in remote places where often Internet access is provided by radio links. It can happen that network on these facilities becomes temporarily isolated from the rest of the world. In order to allow administrator users to access the archive web interface even in these conditions, IA2 maintains a copy of the database, the web portal and its related services on the observatory servers. Since authentication and authorization are centralized, it is not possible to perform the login through external identity providers when Internet access is down. So, a local login page is provided by the portal itself, configured with a single user having admin privileges. Since the portal is able to generate its own tokens, interactions between local version of UserSpace and FileServer are handled with the same logic used by the public version.

FileServer URLs When a user performs a query on the portal, FileServer URLs are displayed as hypertext links inside the portal result table. If files are private, an access token needs to be added to the requests. Tokens are usually sent in request headers, however in this case URLs are put inside HTML anchor tags and it is not possible to set HTTP headers using them. Portal could act as a proxy: it could build an URL referring to itself, which calls the FileServer adding a token in the headers when requested. This can work, however files can be very big, especially for radio telescopes, so being able to access the FileServer directly saves bandwidth. Passing tokens in the URL as query parameters is discouraged because URLs, unlike headers, are stored in server logs and so tokens could be easier stolen by attackers. Best practices suggest to add the JWT id (jti) claim if it is necessary to put tokens in query parameters, in order to avoid token reuse. Validating jti values means that it is necessary to store somewhere a list of already used tokens. Again, if the portal is able to generate its own tokens, it can generate a different token for each private file without exposing the token received from RAP in URLs or without performing additional calls to RAP in order to obtain a new token for each file.

Since tokens expire, it is possible that if the user performs a search on the portal and then don't refresh the page for a long time, hypertext links are not valid anymore. To prevent this possibility, URLs are regenerated when tokens are going to expire and links on the result table are transparently updated performing an AJAX call.

If the portal and the FileServer are hosted on servers sharing the same domain, it may also be possible to decide to store the tokens inside cookies, so they are automatically sent by the browser when clicking on the links. Since it is not possible to set cookies for another domain, putting tokens inside URLs may be necessary when the two services are hosted on different domains.

2.4 Non-browser access

Some users need to access IA2 services like the FileServer from command line. We implemented a simple page reachable from RAP which can be used to generate tokens, providing also the possibility to specify different expiration times. Tokens generated from this interface are not displayed in the HTML page, so that malicious JavaScript code will not be able to read them. Instead, a download is forced setting the HTTP header `Content-Disposition: attachment; filename=token.txt`

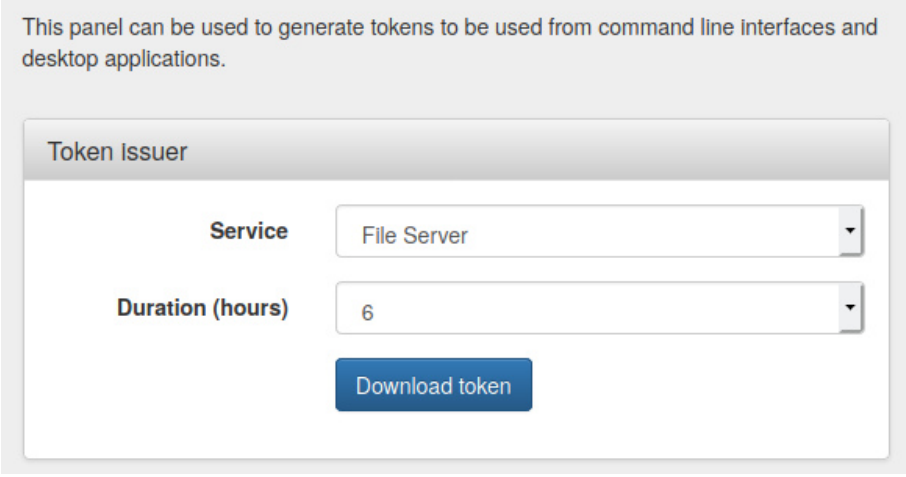


Figure 10: Token issuer

2.5 Apache Guacamole

Apache Guacamole⁵ is a web-based remote desktop gateway. It supports standard protocols like VNC, RDP, and SSH. This means you can use it to access remote machines from your browser.

Guacamole provides an extension for OpenID Connect authentication. ID token is sent using the OAuth2 Implicit Grant Flow^[5]. In the Authorization Code Grant Flow the access token is kept server-side only, while in the Implicit Grant Flow the token can be seen by the browser. This flow is usually used in JavaScript Single Page Applications and it is considered less secure than the Authorization Code Grant Flow, since JavaScript code can manipulate the token directly. In this case token is used only for retrieving the user identity and it is never used again, so the security level is good in any case. Token is returned directly into the redirect URL, however it is not set as a query parameter but in the URL fragment (the part after the hashtag '#'). URL fragments can be read client-side but they are not sent to servers, so the token will never be exposed in server logs.

Guacamole validates JWT signatures retrieving keys from JWKS endpoint. It was necessary to increase the length of the key generated by RAP because Guacamole required a minimum key length bigger than the default value provided by the library used by RAP to generate the key (phpseclib⁶). This demonstrates that, even if OAuth2 is a standard, in order to achieve a real compatibility it is often necessary to depend on specific agreements between applications.

⁵<https://guacamole.apache.org/>

⁶<https://github.com/phpseclib>



2.6 VLKB

ViaLactea KnowledgeBase (VLKB)⁷ is a composition of galactic plane resources (images, data cubes, catalogues, numerical models, ...) and the web interfaces needed to search among and access to them. VLKB services hosted by IA2 have been recently updated in order to work with RAP tokens. This has been done adding a servlet filter which validates the received access token and use it to retrieve groups from the GMS. Also in this case tokens have multiple audiences. SODA engine will also store files on the UserSpace and that call will need a token too.

Often VLKB is accessed using a 3D visualization tool called VisIVO. In order to allow authenticated access this tool needs to implement the OAuth 2.0 for Native Apps specification (RFC 8252)[14].

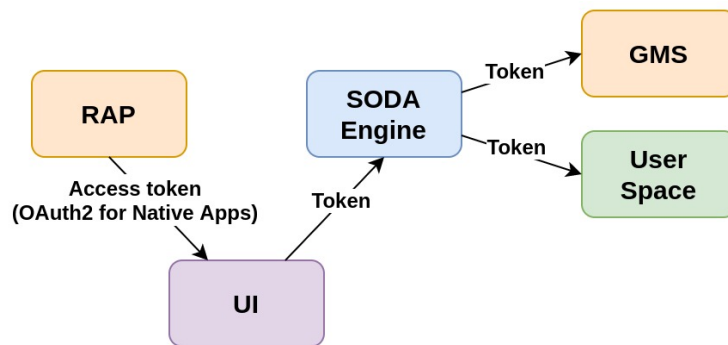


Figure 11: Token usage in VLKB

OAuth 2.0 is built for working in the browser and native applications still need to open a browser in order to retrieve the initial access token. The user accesses the login page in the browser, then there should be a way to send the token from the browser back to the desktop application. One of the most used way consists in making the native application listening on a port and waiting for an HTTP request carrying the token. So, the final OAuth2 redirect references the loopback IP address (localhost), on the port where the native application is listening. The application should also implement a protocol called Proof Key for Code Exchange[15], to avoid that malicious applications running on the same device can steal the token. The application can also obtain a refresh token and use it to obtain new access tokens to keep the user logged in.

3 Conclusions

3.1 Lessons learned

3.1.1 SAML non-browser access is even worse than OAuth2 one

After we joined the eduGAIN federation and before making RAP using standardized OAuth2 calls, we investigated the possibility to provide non-browser access through SAML. We did some tests using a SAML profile called ECP (Enhanced Client or Proxy) which consists in a sequence of SOAP calls between client, SP and IdP. Unfortunately this profile seems to be supported by a very small fraction of the Identity Providers registered in eduGAIN, so it is not feasible to use it. The reason behind this choice is quite obvious: the profile requires the user inserting the credentials directly into the client (that acts like a proxy), so using this profile security gets worse. Moreover, SOAP calls are quite heavy and require a highly customized client. At the end it seems that OAuth2 tokens are easier to use outside the browser than SOAP assertions.

⁷<https://vialactea.iaps.inaf.it/vialactea/eng/index.php>



3.1.2 Account merging is not trivial

Implementing account merging is not trivial as it might seem. First of all the implementer has to decide how to handle the user identifiers involved into the process. Indeed there are two possible alternatives: one of the two identifiers is deleted and the other collects the attributes previously belonged to the former or both the identifiers are deleted and a new identifier is generated, collecting the attributes of both. In both cases there is at least one identifier removed from the system and this information needs to be propagated to the other components. For example, if the authorization information (groups) is managed by a separated service, this service needs to be called in order to move the memberships and privileges from one user (the one that will be deleted) to another. Since this operation is very critical we decided to perform it before deleting the user id, in order to avoid the deletion if an error happens during the transfer. In case of error, a proper rollback procedure should be implemented. Even taking this precaution, concurrency issues could arise (what if another user try to add a permission during a merging process?). Moreover it is possible that the user is logged in on other services (e.g. a portal) during the merge. So it could happen that after a merge some applications have an active session containing an user identifier that doesn't exist anymore. Theoretically, all the services should be aware of the merging feature and proper mechanism for distributing the notification of a merge event should be implemented. In practice, since merge events are not so frequent, a good compromise could be accepting that errors can happen in rare situations, trying to minimize the impact of them in a "best effort" approach.

A completely different implementation could avoid to provide a unique identifier for each user and treat all the users as a list of accounts. In this case the whole list is passed to the services instead of a single identifier. This simplify the merging process, however most of the existing tools expect users to have a unique identifier. Removing this constraint couldn't be feasible, unless modifying the source code of the tool itself and adding additional logic inside the external services (for example parsing the list in order to determine the name to display on a GUI).

Eventually we observed that providing users the possibility to register using a wide range of methods could lead to usability issues and confusions in some of them (it happened that users didn't know what account to use for accessing a service) and this could increase the effort in helpdesk activities.

3.1.3 Think twice before changing A&A

Having a centralized A&A system can bring some advantages in access and identity management, like an homogeneous user experience across all the services.

However such architecture can become a nightmare when it is necessary to change it. Indeed part of the A&A logic is replicated in all the applications. Moreover, modern software development is moving from monolithic approaches towards distributed microservices, increasing the impact of this replication.

Standards like OAuth2 and SAML are quite complex and it takes a lot of time to become familiar with them. Scalability automatically provided by the use of JWT instead of reference tokens wasn't obvious at the beginning and we ended up implementing complex custom solutions in order to replicate user sessions across our services.

Special features needed by astronomical community and rarely considered by industry standards, like command line access, combined with the habit to use simpler authentication methods like BasicAuth, brought us to develop our custom implementations and modify existing tools in order to fit them to our needs.

We based all our services on a first version of the A&A system, which didn't follow the standards properly. When we fixed our implementation we realized that we had to change also all the services which were already connected to it. These services were already in production and since the login was centralized it was not possible to change it in one go without breaking all the existing services. So we needed to set up complex strategies for making the old and the new system coexist for all the transition phase.

3.2 Further developments

Even if we found out strategies for working with tokens from the command line, if we would like to set up long lasting and completely automatic programmatic operations we are still limited by the short token lifespan and by the fact that we need in any case a user performing a login in the initial phase. Having a look at how commercial companies handled these use cases, in particular how Google Cloud Service Accounts work, we



could provide expert users the possibility to generate a keypair and use its private key to sign JWT. Public keys could be exposed through a dedicated JWKS endpoint and users could have the ability to revoke them. Notice that, from a user's perspective, a mechanism like this requires an effort that is similar to managing X.509 client certificates, although the initial setup is lighter because it doesn't need interactions with an external certification authority.

References

- [1] Account linking and LoA elevation use cases and common practices for international research collaboration, <https://aarc-project.eu/wp-content/uploads/2017/03/AARC-JRA1.4H.pdf>, 13th June 2017
- [2] Dummy's guide for the Difference between OAuth Authentication and OpenID <https://nat.sakimura.org/2011/05/15/dummys-guide-for-the-difference-between-oauth-authentication-and-openid/>, 15th May 2011
- [3] Why You Should Always Use Access Tokens to Secure an API <https://auth0.com/blog/why-should-use-accesstokens-to-secure-an-api/>, 11th April 2017
- [4] IVOA Credential Delegation Protocol <http://www.ivoa.net/documents/CredentialDelegation/>, 18th February 2010
- [5] The OAuth 2.0 Authorization Framework <https://tools.ietf.org/html/rfc6749>,
- [6] The OAuth 2.0 Authorization Framework: Bearer Token Usage <https://tools.ietf.org/html/rfc6750>, October 2012
- [7] JSON Web Token (JWT) <https://tools.ietf.org/html/rfc7519>, May 2015
- [8] OAuth 2.0 Token Exchange <https://tools.ietf.org/html/rfc8693>, January 2020
- [9] JSON Web Key (JWK) <https://tools.ietf.org/html/rfc7517>, May 2015
- [10] The Authentication and Authorization INAF Experience, F. Tinarelli, S. Zorba, C. Knapic, G. Jerse, *Astronomical Data Analysis Software and Systems XXVII, ASP Conference Series, Vol. 522*, 2019
- [11] Understanding OAuth2 <http://www.bubblecode.net/en/2016/01/22/understanding-oauth2/>, January 2016
- [12] IA2 authorization and authentication - INAF-OAT Technical Report N. 220, S. Zorba, F. Tinarelli, C. Knapic, G. Jerse, November 2017
- [13] Group Membership Service <http://www.ivoa.net/Documents/GMS/index.html>, February 2020
- [14] OAuth 2.0 for Native Apps <https://tools.ietf.org/html/rfc8252>, October 2017
- [15] Proof Key for Code Exchange by OAuth Public Clients <https://tools.ietf.org/html/rfc7636>, September 2015
- [16] OAuth 2.0 Message Authentication Code (MAC) Tokens <https://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac-05>, January 2014