TIER API Authentication in a Federated World

Universities have thousands of services and clients that must interact with REST APIs. Federated services may have millions of potential clients. How can a client find a service, authenticate, and establish an identity?

It is not feasible for each service to maintain credentials for each potential client.

Resources available from Internet2.

1. Internet2 has a recommended identity solution for people users --- Shibboleth and InCommon metadata.
2. Internet2 has an identity system for services --- InCommon Certificate Service.

User authentication is already well taken care of with Shibboleth. A service can easily identify a user and know something about her (via attributes). Federation allows services to identify users from different institutions. Service authentication is well taken care of with InCommon certificates. A service generally lives at a well know addresses and the certificate verifies ownership of the address.

Most clients do not have this luxury.

Client/Service Registry (SSO for API)

We propose 'SSO' support for API clients, provided by a per-institution registry of services and clients, federated by the InCommon trust model.

A Client/Service Registry (CSR) at each institution, accessible by a standardized API, will provide:

1. General information about each client and service,
2. A chain of trust for each client and service.
3. Maintenance of identity credentials for each entity in the registry (to authenticate to the registry).
4. Temporary credentials for clients to any service in the registry.

It will be seen how easily InCommon's existing resources federate this registry model.

The Client/Service Registry (CSR) Resources

This describes resources and capabilities of the CSR.

Attributes

Vetted attributes provide identity assurance.

Client and Service common attributes:

1. Id: A unique identifier (similar to a user's netid or login id)
2. Name and description: Administrators: People or clients who can administer client information.
3. Contacts: People or groups who can be notified if action must be taken, or to certify the entity is still active.
4. A sponsor: A entity in the registry that certifies the identity of the service or client. Each sponsor also has a sponsor. As an example: client 'chem101a lab' is sponsored by 'Prof. Wimbly', who is in turn sponsored by 'Chemistry', which is sponsored by the root sponsor of the registry. Through this chain of sponsorship the registry can provide identity assurance for its entities.

Additional attributes for Services:

1. Service description URL: What is it, details of its api, who should use it, etc.
2. Service API URL components: Host, port, etc.
3. Supported authentication methods: Authorization service information: if OAuth is supported.

Additional attributes for Clients:

1. Host: if known and constant.
2. Redirection URLs: If OAuth is supported.

In addition the CSR maintains a long-term authentication credential to itself for each client in its registry. A service may authenticate to the CSR with its InCommon certificate or a maintained credential.

Credentialing

Temporary credentials allow any client in the federation to securely contact any service in the federation.

This describes the means by which a client gets a temporary credential and how a services verifies it.

- Details depend on the type of credential.
- 'Temporary' means the credential can be easily renewed. It does not specify any limits on the validity duration.
- The registry neither prescribes nor proscribes any particular authentication method.

Note that for intra-institutional access temporary credentials may not be required. If a client id is handled similarly to a user id the institution's credentialing system, e.g. kerberos, may handle credentialing external to this specification.

Basic authentication with password:

1. The client requests a password from the CSR:
    a. POST (CSR)/Token?type=password&service=(service_id)
2. The CSR generates a password (an opaque string, valid only at the target service) and returns it to the client, along with expiry information.
3. The client accesses the service, using "Authorization: Basic <client_id:password>"
4. The service asks the CSR to verify the password:
    a. GET (CSR)/Client/(client_id)?token=(the password)
5. If the password is valid the CSR returns the client's resource record, along with expiry information.
6. Both client and service may cache the password for its lifetime.

Certificate authentication:

1. The client generates a certificate request and requests a certificate from the CSR
    a. POST (CSR)/Token?type=certificate (the PEM certificate request is in posted content)
2. The CSR, acting as a simple CA, signs the request and returns the signed certificate it to the client.
3. The client accesses the service, using client certificate authentication. Its id and the certificate expiry are contained in the certificate.
4. The service verifies that the certificate was issued by the CSR, using its copy of the CSR's CA public key, and that it has not expired.
5. For its lifetime the certificate may be used with any service in the domain of the CSR. The CSR does not maintain a CRL.

Basic Authentication with JSON Web Token (JWT):

1. The client requests a JWT from the CSR:
    a. POST (CSR)/Token?type=jwt&service=(service_id)
2. The CSR generates a token, signs it with its own InCommon private key, and returns it to the client.
    a. issuer: the CSR
    b. subject: the client
    c. target: the service
    d. expiration: some time
3. The client accesses the service, using "Authorization: Basic <the token>"
4. The service verifies that the token was signed by the CSR, extracts the client_id, and queries the CSR for information:
    a. GET (CSR)/Client/(client_id)
5. The CSR returns the client's registry information.
6. Until its expiration the token may be reused by the client to the service.

Security Considerations

Security of client and server certificates

TLS is sufficiently mature that its security concerns are well know and much discussed. Clearly a certificate authority, in this case Comodo and the various CSRs, must guard their private keys.

Security of password authentication

The password is a bearer token, so the client must guard it from disclosure.

The most common attacks against password authentication are phishing, guessing,and reuse.

- An API client will not normally be susceptible to phishing.
- If the password is sufficiently long (e.g. 128 bits) it will not be guessable.
- The password is valid only for the specific client to the specific service. It cannot be reused for to service.

Security of JWT authentication

The JWT is a bearer token, so the client must guard it from disclosure.

Federated Identity

Federation is easily accomplished because the institutional registries can know each other through the InCommon Registry of CSRs, and can mutually authenticate with InCommon certificates.
A client can, through the link between its own CSR and a remote service's CSR, obtain a credential to the remote service. The service can, in turn, verify the credential at its own CSR and lookup the client through the same CSR to CSR connection. The CSR-to-CSR link can be seen as a no-cost, maintenance-free portal, through which any client can acquire a credential to any service in the federation.

Example scenarios

1) Federated access using Basic client authentication to the service

The scenario involves a collaboration between the University of Washington and Old Dominion University.

Definitions

- CSR: Client/Service Registry
- UW: University of Washington
- ODU: Old Dominion University
- CSR.UW: UW's CSR
- CSR.ODU: ODU's CSR
- GWS: Group service at UW

The problem

Students at UW and ODU are working on a project. As part of their collaboration an application at ODU needs access to groups maintained by a student (uwuser) at UW. The application is a registered client with the CSR.ODU and has an id of 's_ourapp@odu.edu'.
The group service is a registered service at CSR.UW with an id of 's_gws@washington.edu'.

The client and service interact through the TIER REST API for groups.

The flow

(Assume a central registry of CSRs maintained by InCommon.)

1. 'uwuser' adds 'c_ourapp@odu.edu' as a member manager of the groups she wishes to share.
2. When the application needs to works with her groups it asks CSR.ODU for an authentication token to GWS.
    a. POST (CSR.ODU)/Token/?service=s_gws@washington.edu?type=basic
    b. CSR.ODU relays the request to CSR.UW
        i. POST (CSR.UW)/Token/?service=s_gws@washington.edu&client=s_ourappi@odu.edu&type=basic
        ii. CSR.UW responds with a token for s_ourapp@odu.edu to access s_gws@washington.edu.
    c. CSR.ODU relays the response, a Basic Authentication token, to the application.
3. The client manages uwuser's group memberships through GWS's REST API, using the token as Basic authentication.
    a. Since the token was issued by UW's CSR it is easy for the group service to verify it.
        i. GET (CSR.UW)/Client/s_ourapp@odu.edu?token=(the password token)

Notes on the flow

1. The application could have requested other forms of authentication, client certificate or JWT, in step 2.

2) Federated OAuth2 (authorization code grant) access using Basic client authentication

Definitions

- CSR: Client/Service Registry
- UW: University of Washington
- ODU: Old Dominion University
- CSR.UW: UW's CSR
- CSR.ODU: ODU's CSR
- MMUI: Member manager application at ODU
- GWS: Group service at UW
- AS-GWS: GWS's authorization server

The Problem

While engaged in the collaboration with students at ODU, 'uwuser', a student at UW, became fond of a general purpose group member manager GUI (MMUI) at ODU. She now wants to use that application with some of her GWS groups in a new collaboration at UW. This time she will manage the group memberships herself, using the MMUI application.

The flow

1. 'uwuser' accesses MMUI with her browser.
2. MMUI asks what groups she wants to manage.
3. From her previous collaboration she knows the id of UW's group service (s_gws@washington.edu) and enters it, along with the names of the groups she wants to manage:
    a. u:users:uwuser:chem401b:*
    b. u:users:uwuser:chem401b-lab:*
4. MMUI asks CSR.ODU for information about GWS
    a. GET (CSR.ODU)/Service/s_gws@washington.edu
    b. CSR.ODU relays the request to CSR.UW
        i. GET (CSR.UW)/Service/s_gws@washington.edu
    c. CSR.ODU relays the response to MMUI
5. From the response MMUI knows GWS supports OAuth and learns its authorization server's endpoint. MMUI redirects uwuser to AS-GWS with:
    a. client_id: s_mmui@odu.edu
    b. scope: u:users:uwuser:chem401b:*(member-manage) u:users:uwuser:chem401b:*(member-manage)

    c. redirect_uri: (MMUI)/oauth_code/
6. AS-GWS does not have an existing relationship with MMUI, so it looks up MMUI by its id.
    a. GET (CSR@UW)/Client/s_mmui@odu.edu
    b. CSR.UW relays the request to CSR.ODU
        i. GET (CSR.ODU)/Client/s_mmui@odu.edu
    c. CSR.UW relays the response to AS-GWS
7. AS-GWS asks uwuser:
    a. Application "The Old Dominion Fabulous Member Manager" wants permission to manage the memberships of your groups:
        i. All groups on the stem: "People on my Chem 401b projects"
        ii. All groups on the stem: "People in my Chem401b lab"
8. uwuser asks for 'more' info.
9. AS-GWS queries CSR.ODU for sponsorship information, via CSR.UW, and reports to uwuser, "MMUI is certified by ODU IT, which is certified by Old Dominion University, which is certified by InCommon."
10. uwuser clicks 'OK'
11. AS-GWS redirects uwuser to MMUI's redirection URL (verified in step 6), with an authorization code.
12. MMUI needs to convert the code into an access token. It asks CSR@ODU for an authentication token for AS-GWS.
    a. POST (CSR.ODU)/Token/?service=s_gws@washington.edu&type=basic
    b. CSR.ODU relays the request to CSR.UW
        i. POST (CSR.UW)/Token/?service=s_gws@washington.edu&client=s_mmui@odu.edu&type=basic
    c. CSR.ODU relays the response, a Basic Authentication token, to MMUI.
13. MMUI requests an access token from GWS
    a. POST (AS-GWS)/Token
    b. with the authorization code in posted parameters
    c. and using the just received password for authentication
14. AS-GWS verifies the password
    a. GET (CSR.UW)/Client/s_mmui@odu.edu?token=(the password from step 12)
15. AS-GWS responds to MMUI's request with the access token.
16. MMUI uses the access token to manage uwuser's group memberships through GWS's REST TIER API.

Notes on the flow

1. If uwuser had not known the identity of GWS she could have asked MMUI to search for the service at UW (via CSR.ODU and CSR.UW)
2. AS-GWS could cache the information it received about MMUI for some time (TBD), but CSR.ODU will remain the official SoR for that information.
3. MMUI could have requested other forms of authentication, client certificate or JWT, in step 12.
4. In none of this was any action or authorization required of the administrators of CSR.ODU, CSR.UW, or GWS.

Notes

Relationship of CSR to OAuth2

This proposal is not OAuth2. This is about authentication. OAuth2 is not an authentication protocol.

This does, however, support OAuth2.

OAuth2 requires one client authentication—client to authorization service. But it does not specify an authentication mechanism, nor the registration of the client, nor maintenance of the client's credential. These are left up to the deployer. There is a suggested method for client self-registration (RFC-7591), but it uses entirely client asserted information—not what we need.

Question: What about https://tools.ietf.org/html/rfc7591? Specifically, why not use signed software statements to specify client registration behavior? (page 18)

1. That protocol addresses a different need, with different resources.
    a. The goal of 7591 is a client registered with an authorization service. No more than that. The relationship is, afterwards, an isolated couple. The client has a unique id and password with that authorization service. It has information stored there. But there seem to be no means to edit, expire, or delete that client data. Whatever small bit of 'federation' provided by even a signed software statement has been lost.
    b. It would seem that 7591 is trying to make it as easy as possible for resource providers to acquire clients.
    c. A problem OAuth2 has, IMHO, is that it always wants to find solutions entirely within itself. We have other resources: Internet2 and the InCommon Federation. We have other goals: federation and SSO for API.
2. If we added signed statements to the API, it's not clear how a service would verify the document - that was signed by a remote CSR.

Further Question: Why not use Roland Hedberg's RFC draft OpenIDConnect for Federations? (you can RFC-ify that at: https://xml2rfc.tools.ietf.org/)

1. Roland's proposal does add a bit of federation to rfc-7591. Some other random thoughts:
    a. Proposals that start with "TLS can't be trusted" seem disingenuous. The internet runs on TLS. If it's broken it gets fixed. This is especially annoying when people are talking about OAuth2, which pushed out old OAuth chiefly because OAuth2 used and trusted TLS.
    b. It is still contained entirely within OAuth2. It makes no use of existing resources provided by the InCommon federation.
    c. It doesn't address the id and credentialing issues.

CSR's use of PKI

OAuth2 so totally abhors client certificate authentication that one wonders if its creators had bad experiences with PKI as children. We feel that client certificates, when used for very easy and well supported holder-of-key authentication, need to be included in the list of possible authentication methods.

Relationship to RFC 7521

This was an attempt from a couple of years ago to use OAuth2 assertions for client authentication. It seems forced and cumbersome, the very epitome of, "If you have a hammer everything looks like a nail."

OAuth2 is not an authentication protocol.