Fig. 1

1. Toward Interoperable Auth

Markus Demleitner
msdemlei@ari.uni-heidelberg.de

- What's the point of this?
- How astroquery/pyVO auths to Gaia, CADC, LSST
- How could this come together?

(cf. Fig. 1)

2. Where to?

- Auth is a fact of life. It even has ethical uses! I, for instance, would like to have persistent table uploads in DaCHS, but as long as that requires DaCHS operators to do user management, that's not worth the effort.
- We can only expect clients (TOPCAT, pyVO, ...) to support auth if it's halfway clear what they need to implement.
- Expecting extra code per service is anti-VO.
- So: Can we, based on current services, figure out rules and a metadata schema so one code rules them all?

Spoiler: No. (But wait)

3. Gaia Auth

1. Get the TAP access URL, append `/login` to it.
2. Post formencoded username/password to it
3. Fiddle out JSESSIONID from the response's Set-cookie header
4. This token is passed to the service in a JSESSIONID cookie.

Gaia folks in the audience: Correct me now if I'm wrong.

4. CADC Auth

1. Fetch the services capabilities.
2. Pull out an access URL for a capability with standardID `ivo://ivoa.net/std/UMS#login-0.1`
3. Post formencoded user/password there
4. Pull the token out of the response body
5. This token is passed to the service in a CADC_SSO cookie (modulo a bit of syntax, perhaps).

CADC folks in the audience: Correct me now if I'm wrong.

5. LSST auth

1. Go to cilogin.org in a fat web browser, perform some actions that can't reasonably be automated.
2. Get a token out.
3. This token is passed in an RFC 6750 Authorization: Bearer header.

LSST folks in the audience: Correct me now if I'm wrong.

6. Differences and Parallels

- All use http request headers: Gaia and CADC `Cookie:` with some custom name, LSST RFC 6750 authorization.
- Gaia and CADC post username and password to a discoverable URL, LSST needs prior knowledge and manual labour.
- Gaia uses a TAP 1.0-style fixed endpoint name for the token service. CADC has an extra capability.

7. What if?

Let's assume for a second all three agree to use RFC 6750 ("Authorization: Bearer"). That's perhaps not super-attractive to the services that so far use cookies, because those naturally work with web browsers. But then I don't think there's a major overhead to accept credentials in either Cookie or Authorization headers.

All a client would then need to know to run such a service:

Where do I get the token?

This would include "from a kind human", "from a service accepting username and password", and probably more in the future.

8. A First Proposal

I claim the token generator needs to be per security method (you'll want a different one for RFC 6750 than for client certs, say). Hence we should – perhaps as annex to SSO? – define a schema that defines a type `sso:RFC6750Auth` (or do we claim it's OAuth all the way? Experts, speak out!), and our services would then say:

```
<!-- Gaia: -->
<securityMethod standardID="ivo://ivoa.net/sso#RFC6750">
  <tokenGetter type="userpass">https://<gaia>/tap/login
  </tokenGetter></securityMethod>

<!-- CADC: -->
<securityMethod standardID="ivo://ivoa.net/sso#RFC6750">
  <tokenGetter type="userpass">https://<cadc>/anywhere/login
  </tokenGetter></securityMethod>

<!-- LSST: -->
<securityMethod standardID="ivo://ivoa.net/sso#RFC6750">
  <tokenGetter type="manual">https://cilogin.org
  </tokenGetter></securityMethod>
```

The userpass type would essentially tell a client: "Prompt the user for a conventional pair of username and password, and POST these as application/x-www-form-urlencoded with keys username and password (that's what Gaia and CADC have independently come up with anyway).

Where Gaia and CADC don't agree is whether the token is returned in a set-cookie header (advantage: you can use exactly what you'll use with a plain web browser) or in the payload (advantage: doesn't abuse set-cookie). Since I think with minimal conventions (the cookie name) set-cookie works well enough, my take would be to accept a certain amount of spec seediness and go for set-cookie.

Then there's the interesting question of whether securityMethod:tokenGetter should be 1:1 or 1:n. 1:1 would make life easier for clients (because they don't have to decide anything), but 1:n looks about right for either "I accept tokens from any of these different operators" or "Have your choice between manual, userpass and perhaps something else". Do we need to tell the cases apart? Should we have an @title so clients can leave the choice to the user?

9. Then:

- A client like TOPCAT could do all auth, including obtaining the credentials and turning them into a token, for Gaia and CADC.
- For LSST, it could at least open a web browser and ask for the token.

I'd say that'd be a reasonable start.

10. Of course:

- This still doesn't (automatically) give us federated auth...
- ... nor credential delegation.

We could probably tackle at least federated auth by requiring the token to be a piece of signed (encoded) json.

I would just love if I could point to a different tokenGetter and had a way to check its tokens...

Thanks!

11. Bonus: RFC 8252

Turns out there's an RFC that talks about obtaining OAuth2 tokens from within actual programs: RFC 8252.

It essentially pokes holes into the browser sandbox claiming that's more secure (because the actual program doesn't see the credentials).

Either I don't understand the reasoning – or we want to ignore RFC 8252.