AAI for CMS Data

Brian Bockelman

Desired CMS data access policies (simplified)

- CMS *logical* namespace starts at `/store`.
- Only CMS users may files inside `/store`.
- Users can write “permanent” outputs to `/store/user/$USERNAME` at sites they are assigned to.
  - Only the “owner” allowed to write/create/delete files inside their own `/store/user` area.
  - Never developed a collaboration-wide policy about file ownership after user leaves.
- Users can write transient outputs to `/store/temp/user/$USERNAME.$ID_HASH` at *any* CMS site.
- Only `/cms/Role=production` is allowed to write in the remainder of CMS namespace.

User Data Flows

- User jobs read via ROOT using a custom set of CMSSW plugins. About 20% of user jobs stream data from offsite (via XRootD).

- User jobs write via a custom set of plugins. The plugin to use is controlled by the site or processing framework (CRAB3).
    - The most popular plugins are based on `gfal-copy` and `xrdcp`.

- User stageout goes to:
    - Local site storage. Failing that,
    - “Fallback site storage”. Each site configuration specifies zero or one “fallback” output area. Failing that,
    - “Home storage”. Ultimate destination for the user outputs

User AAI

- User identity credentials (limited X509 user proxy) are shipped to the worker node:
  - Reads are done with user proxy, initiated by job wrapper.
  - Writes are done with user proxy, initiated by job wrapper.
  - Copies from transient to permanent storage is done by a central component (ASO). User delegates proxy to MyProxy and allows ASO to receive a copy.
    - Hence final transfer is done with the user credentials.
- Traceability is “trivial” - everything is done under the user’s identity, meaning sites ought to do this for us.

Works Well
Except when it doesn't

- This setup is about as "traditional grid" as possible. Downsides are well-known:

  - Storage considers the files to belong to the user, *not* the VO.

  - CMS cannot separately manage user files. CMS provides *no* tools for managing user storage, either for sites or users.

  - User credentials fly all across the world; they are quite powerful! *Every single job has the power to delete files from disk.*

Random Thoughts

- We don't want to ship identities to worker nodes. We probably want to ship access tokens:
  - "The bearer of this token is allowed to write into `/store/user/bbockelm`. Signed, CMS"
  - Access tokens are well understood by both HTTP and XRootD.
  - This is basically the "ALICE model".
- Google's libmacaroons defines a mechanism for this. These provide:
  - **Decentralized verification**: Don't need to call back to the issuing service to validate.
  - **Attribute / value pairs**: We can build our own schema on top of these.
  - **Attenuation**: I can generate a new token with additional restrictions before handing it to the worker node. E.g., **limit token to only writing in a specific directory at a list of sites**.
- Tokens are passed through a standard HTTP header. Trivial to use with `curl` or `xrdcp`. Contrast with X509 certificates, which must work with transport layer.
- Macaroons could be used to protect privacy - says *what* bearer can do, not *who* they are.
  - Traceability would become a joint responsibility of site and VO.

Potential Demonstrator

- I'm interested in a demonstrator showing:
  - Based on user identity, macaroon issued by a VO specifying read/write permissions.
  - Token delegated to a job, where it is restricted to just write that job's output.
    - Using a standard tool (curl, xrdcp, gfal-copy), stageout occurs with token.
    - Storage stores file as belonging to "CMS" but creates an audit trail to the original token.
- Since ALICE has previously done something similar with XRootD, the xrootd daemon actually has most of the plugin APIs needed for such a demonstrator.
  - Since tokens are so common in HTTP frameworks, I suspect these also have the appropriate plugins.