Rob Carter thoughts on fine-grained authZ on APIs for data access

Email of 17 June 2015

This isn't the entirety of what I was hoping to find, and it's actually somewhat earlier in the overall discussion we had locally about this stuff, but it gets some of the basic ideas out on the table, at least, and comes with the obligatory boxes-and-arrows diagram. A couple additional background notes to frame up what's below:

(1) In service of trying to prove that all problems (save one :-) can be solved by the imposition of an additional layer of indirection, we came up with the idea of interposing "reflection APIs" between common systems of record and common data provisioning targets that would serve to encapsulate the source /target specific APIs and fold them into a somewhat standard API framework that we could easily code against. The idea was that then if we ran into something like a change in the API from, say, Grouper, or when we finally succeed in replacing OIM with something else (a TIER registry, perhaps) we'd only have to adjust the RAPI code — everything that talks to Grouper (or OIM, etc.) would be rewritten once (to talk to the RAPI) and then the RAPI would be the sole point of change to adjust to changes at the endpoints.

(2) We realized along the way that, at least in our environment, there are really *4* relevant elements in the tuple that describes an attribute permission — rather than the typical (subject,action,resource), we really care about (subject,action,resource,context) where the "context" is usually the referent of the resource. So, for example, it's not enough to say that "doctors in Pulmonology can read tidal volume test results in the EHR" — we want to instead be able to say that "doctors in Pulmonology can read tidal volume test results in the EHR for current patients in Pulmonology", or in an academic sense, we want to be able to say that "department chairs can read salary information out of the faculty management system for faculty in their departments" rather than simply "chairs can read faculty salary information". A lot of back-and-forth went on about modeling that 4-tuple in the Grouper 3-tuple environment — whether it was more efficient/effective/"correct" to merge the action and resource into a new action and store the context in the resource slot (e.g., User can Read-Sensitive-Data on Students) or whether it would work better to merge the resource and context (User can Read Student-Sensitive-Data). I think we eventually determined that the former was more tractable, mostly because we have a fixed (albeit large) number of attributes in play (around 400-450) and a much larger and growing number of contexts (possibly one for each of 800,000 potential users and each of roughly the same number of groups). It makes the logic somewhat harder to grasp, I think, but just in terms of scaling, it's easier to scale to a fixed 400-500 items than an increasing 1.6 million :-)

(3) At the time we were talking about this, informed consent began and ended with uApprove extensions in the IDP, so our ideas about user consent were all focused on the Oauth service we were working on bringing up for use with some student information for student-driven mobile app development. The Oauth bit at the end of the notes below is the late 2013 version, in a way, of the idea of an UMA implementation and some form of pass-thru via the authorization API to the consent store.

If I can get over to the whiteboard later, I'll try and send you some additional bits from that, but I think at a mid-level approximation, this kinda frames what it is we've been discussing off and on for a couple years in this space. Some of the implementation details continue to drift in response to things like TIER and the PrivacyLens work, but all in all, the grand scheme seems to survive the discussions. Whether it can survive any sort of implementation is a different (and as yet unanswered) question :-).

—Take care!—
—Rob—

Email of 30 September 2013

Here are the stipulations I had jotted down (rewritten somewhat in light of our later conversations) regarding the APIs we've been discussing. Attached below that is a Graffle version of the diagram we built last week... I'm passing it along in PDF for those of you who can't handle Graffle docs, and Graffle for those who might want to make edits directly.

--Rob--

=========
Assumptions:

* The reflection APIs (or RAPIs) are designed as web service exposures of the native APIs associated with the back-end systems they're attached to. They are not caller-specific (that is, reflection APIs are "in the context" of the back-end systems they wrap).

* The RAPIs are a virtual representation of the back-ends, not a direct one-to-one mapping. They reflect the way we'd *like* the back-end APIs to be.

* There will be no (or nearly no) attribute-specific methods in the RAPIs -- methods may take attribute names as arguments, but are not themselves attribute-specific.

* All access to the Reflection APIs is authenticated -- there will be no anonymous access allowed.

* All calls in the RAPIs will make calls to the AuthZ api before returning results. No results will be returned without authorization validation.

* All calls in the attribute/object web service will likewise perform authorization before returning results or performing actions.

* Authorization calls will typically be 4-tuples of the form "Can <authn'd user> perform <requested operation> on <data element type> about <target entity>. Data element type and operation may be merged into a single 2-tuple for the sake of processing (eg., "can rob perform update-sensitive-data on rb186?").

* Model applications in two ways -- if the application is a transparent proxy for the RAPI or the attribute/object API, it should proxyAuth via a token as the calling user; If the app is performing complicated operations and high-level authorization in its own code, it should authenticate as a service account (possibly with admin rights). In the former case, the PEP is in the API -- in the latter case, the PEP is split between the API and the caller or higher-level WS).

* Some logic may be shared or repeated between callers of the APIs, or within the APIs themselves. When this is determined to be the case, the relevant logic should be externalized in a common routines WS.

* Caller-specific APIs wrap the attribute/object web service in order to present higher-level methods in the context of specific callers (PS, SAP). Caller-specific APIs are created as needed either to simplify the use of the attribute/object API by non-IDM groups (where appropriate) or to enforce internal workflow for operations instigated by those callers (eg., instead of SAP calling the "create user" object API method, it might call an "sapNewEmployee" routine that wraps the "create user" method with various authorization, validation, etc. checks and that performs specific business logic limiting the ability of the SAP caller to, eg., set eduPersonAffiliation=student).

* The Authorization WS acts as a generic PDP wrapper API. We anticipate that the majority of the heavy lifting associated with the PDP function can be performed within Grouper, itself, with the AuthZ WS acting as a marshaling and refactoring wrapper around Grouper's authorization mechanism.

* If necessary, integration with an OAuth Token Broker could be done through the authorization API -- the AuthZ api could, eg., perform a Grouper check to determine if a particular operation should be allowed, returning acceptance if Grouper returns acceptance, but performing a secondary check with the Token Broker for a user-to-user explicit access grant if a token is presented and Grouper denies the operation.

—Rob—

See Also:

TIER Working Groups Home