An Introduction to User-Managed Access (UMA)

Eve Maler
VP Innovation & Emerging Technology
eve.maler@forgerock.com
@xmlgrrl

October 28, 2014
Challenges in apps that handle personal data and content
Some apps are still in the Web 1.0 dark ages

- Provisioning user data by hand
- Provisioning it by value
- Oversharing
- Lying!
Some other apps are still in the Web 2.0 dark ages

- The “password anti-pattern” – a third party impersonates the user
- It’s a shared secret honeypot
- It’s a gray-market B2B partner
Apps using OAuth and OpenID Connect hint at a better, if not perfect, way
What about selective person-to-person sharing?
Our choices: send a private URL...

- Handy but insecure
- Unsuitable for really sensitive data
Import Fidelity Tax Information Into TurboTax®

If you are a Fidelity customer and use TurboTax®, you may be able to import certain information directly from your account into the software. Here’s how.

How to import your information

Once you receive your 1099 statement by mail or through eDelivery, have it available to verify the imported information. Follow these simple steps:

1. Enter your Social Security number (SSN), taxpayer identification number (TIN), or username, and then your password. When asked where to import information from, select Fidelity Investments and enter the same information that you use to log on to Fidelity.com. Then, the tax information available for each of the accounts associated with your SSN should appear.
...or implement a proprietary access management system
Killing – or even *wounding* – the password kills impersonation.
IoT 2.0 is here – and it too needs authorization
We have tough requirements for delegated authorization

- Lightweight for developers
- Robustly secure
- Privacy-enhancing
- Internet-scalable
- Multi-party
- Enables end-user convenience
Introducing UMA
UMA in a nutshell

- It’s a draft standard for “authorization V.next”
- It’s a profile and application of OAuth V2.0
- It’s a set of authorization, privacy, and consent APIs
- It’s a Work Group of the Kantara Initiative
- It’s not an “XACML killer”
- Founder, chair, and “chief UMAntarian”:
- It’s heading to V1.0 in Q1 2015
The new Venn of access control
UMA turns online sharing into a Privacy-by-Design solution

The “user” in User-Managed Access (UMA)

Alice’s authorization service hears Bob knocking on the resource – do the rules say he can come in?
The UMA protocol enables key new use-case options

I want to share this stuff selectively
- Among my own apps
- With family and friends
- With organizations

I want to protect this stuff from being seen by everyone in the world

I want to control access proactively, not just feel forced to consent over and over
UMA is about interoperable, RESTful authorization-as-a-service

Outsources protection to a centralizable authorization server

Has standardized APIs for privacy and “selective sharing”
Use-case scenario domains

- Health
- Financial
- Education
- Personal
- Government
- Media
- Behavioral

Technologies:
- Web
- Mobile
- API
- IoT
UMA-enabled systems can respect policies such as...

Only let my tax preparer with email TP1234@gmail.com and using client app TaxThis access my bank account data if they have authenticated strongly, and not after tax season is over.

Let my health aggregation app, my doctor’s office client app, and the client for my husband’s employer’s insurance plan (which covers me) get access to my wifi-enabled scale API and my fitness wearable API to read the results they generate.

When a person driving a vehicle with an unknown ID comes into contact with my Solar Freakin’ Driveway, alert me and require my access approval.
The user experience can simulate OAuth or proprietary sharing paradigms, or even be invisible (“better than OAuth”)
Under the hood, it’s “OAuth++”

Loosely coupled to enable an AS to onboard multiple RS’s, residing in any security domains

This concept is new, to enable person-to-person sharing driven by RO policy vs. runtime consent
The RS exposes whatever value-add API it wants, protected by an AS.

The RPT is the main “access token” and (by default – it’s profilable) is associated with time-limited, scoped permissions.
The AS exposes an UMA-standardized protection API to the RS

The PAT protects the API and binds the RO, RS, and AS
The AS exposes an UMA-standardized authorization API to the client.

The AAT protects the API and binds the RqP, client, and AS.

The client may be told: “need_claims”
The AS can collect requesting party claims to assess policy

A "claims-aware" client can proactively push an OpenID Connect ID token, a SAML assertion, a SCIM record, or other available user data to the AS per the access federation’s trust framework.

A "claims-unaware" client can, at minimum, redirect the requesting party to the AS to log in, press an “I Agree” button, fill in a form, follow a NASCAR for federated login, etc.
UMA enables business logic centralization, even for “classic” access management

**Business SaaS SSO today:**
- Company X contracts with Salesforce.com
- Employees SSO in from web or native app, passing in role/group attributes
- Company X’s policies at SFDC govern what features users can access
- Company Y does the same at SFDC, etc.
- Company X does the same at Concur, etc.

**Central authz tomorrow:**
- Company X runs an UMA AS
- SFDC’s UMA RS onboards to that AS and respects UMA tokens issued by it, containing entitlements based on Company X’s policies
- Company X’s keeps central policies for SFDC, Concur, etc. (authoritative “AzP” respected each “AzRP”)
- Company Y keeps central policies for SFDC, Concur, etc. (a different authoritative “AzP” respected by each “AzRP”)
The UMA consent model supports robustly partitioned rights and obligations.

These underlie "access federation" trust frameworks.

Akin to...

Identity federation trust frameworks.
Conclusion and next steps
Thank you!

Eve Maler
VP Innovation & Emerging Technology
eve.maler@forgerock.com
@xmlgrrl
Appendix:
The gory UMA details
RS needs OAuth client credentials at AS to get PAT
C needs OAuth client credentials at AS to get AAT
All protection API calls must carry PAT
All authorization API calls must carry AAT

1. RS registers resource sets and scopes (ongoing – CRUD API calls)
2. C requests resource (provisioned out of band; must be unique to RO)
3. RS registers permission (resource set and scope) for attempted access
4. AS returns permission ticket
5. RS returns error 403 with as_uri and permission ticket
6. C requests authz data, providing permission ticket
7. (After claims-gathering flows not shown) AS gives RPT and authz data
8. C requests resource with RPT
9. RS introspects RPT at AS (if using default “bearer” RPT profile)
10. AS returns token status
11. RS returns 20x
Tokens and the tuples they represent