UMA: Enabling New Access Control Solutions for People, Things, and Services

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

March 5, 2015
Personal data sharing challenges
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 honeypot for shared secrets
- B2B partners are in the “gray market”
Apps using OAuth and OpenID Connect hint at a better (if not perfect) way
What about selective party-to-party sharing?
Our choices: send a private URL...

- Handy but insecure
- Unsuitable for really sensitive data
...or require impersonation...

**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
Solution space
The new “Venn” of access control

OpenID Connect

identity federations

privacy individuals

security institutions

OAuth 2.0

UMA
**UMA in a nutshell**

- It’s a protocol 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 two V1.0 Draft Recommendation specs
- It’s not an “XACML killer”
- Founder, chair, and “chief UMAntitarian”:
UMA standardization in context

- **OAuth 1.0, 1.0a**
- **WRAP**
- **OAuth 2.0**
- **Dynamic Client Reg** (from UMA/OIDC contributions)
- **JWT**

- **OpenID AB/Connect**
- **OpenID Connect**
- **UMA Core, OAuth Resource Set Registration**

Tinyurl.com/umacore & oauthrsr are Kantara Initiative Draft Recommendations undergoing All Member Ballot

Interop testing under way
UMA-enabled systems can respect policies such as...

Only let my tax preparer with ID 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 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.
UMA Binding Obligations

- Distributed authorization across domains? Scary!
- This “legal” spec enables parties operating and using software entities (and devices) to distribute rights and obligations fairly in *access federation* trust frameworks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual or Non-person entity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Important state changes when new pairwise obligations tend to appear:
- Token issuance
- Token status checks
- Permission registration
- Claims gathering
- Access requests
- Successful access
A word about the “OpenUMA” community

OpenUMA

You know that blue “Share” button in Google Apps? Ever wanted to add a feature like that to your own app or API ecosystem? The UMA protocol enables you to do just that.

User-Managed Access (UMA) is an OAuth-based protocol that enables an individual to control the authorization of data sharing and service access made by others.

The OpenUMA community shares an interest in informing, improving, and extending the development of UMA-compatible open-source software as part of ForgeRock’s Open Identity Stack. Currently no open-source OpenUMA code has yet been published, but keep an eye out in early 2015!

On this page:

- About the UMA Standard
- Project goals
- Sample Use Case
- Infographic: UMA
- The OpenUMA video
- OpenUMA blog posts
- Get involved!

Leaderboard

#1 Peter Major 405
#2 Victor Ake 398
#3 Brad Tumy 350
#4 Scott Heger 342
#5 David G. Simmons 341
#14 Eve Maler 147

The leaderboard is based on our rockin’ informal points system, read about it here.

Recently Active Members

THE UMA STANDARD EXPLAINED
What can the UMA experience be like?
Architecture
Authorization services architecture for low-scale environments

Monolithic resource owner

Application (Requesting client)

PEP (Policy Enforcement Point)

Request authorization

PDP (Policy Decision Point)

Perm, Deny NotApplicable Indeterminate

Centralized Policy Enforcement

PAP (Policy Administration Point)

Monolithic resource owner

No automated mechanism for onboarding partners

Monolithic resource owner

Enforcement points mediate all interactions

Decisions are fine-grained but “cooked down”

Authorization Policies

soap
xml

Monolithic resource owner
Why an OAuth-based architecture instead?

Diagram:
- (A) Authorization Request → Resource Owner
- (B) Authorization Grant
- (C) Authorization Grant → Authorization Server
- (D) Access Token
- (E) Access Token → Resource Server
- (F) Protected Resource
“User-centric, constrained, delegated, RESTful WS-Security” 😊

It’s about more than “eliminating the password anti-pattern”
- Gets client apps out of the business of storing passwords
- “Teaches” clients to rotate secrets
- Friendly to authentication methods and user devices
- Allows per-client tracking and revocation
- Allows for scoped access
- Captures user consent
- Lowers development cost
- Generative for a wider variety of design patterns
Some interesting properties of this authorization services architecture that contribute to higher scale

- Application (Requesting client)
- Cloud/On-prem Authorization Server
- Resource Server
- Policy Administration (Scope/Consent Administration Point)
- Resource Administration

Decision point hands out entitlements...

Protect requested Scope Entitlements returned Tokens Consent

Local entitlement enforcement

...directly to client apps

Manage

Requesting party

“Consent” to release

Resource Owner

Manage

Resource owner is prepared to be an individual
Now we come to UMA

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"
- "authz relying party" (AzRP)
- "authz provider" (AzP)
- Identity provider (IdP)
- SSO relying party (RP)
- Resource owner
- Consent
- Control
- Negotiate
- Manage
- Requesting party
- Client
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 party-to-party sharing driven by RO policy vs. run-time 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 profileable) is associated with time-limited, scoped permissions.
Comparing plain OAuth access tokens to UMA RPTs

- OAuth access tokens:
  - Profilable; no standardized form on the wire, though a signed JWT is sometimes used
  - Token introspection at runtime is getting standardized; a JWT gets returned

- UMA RPTs:
  - Profilable; default form on the wire (“bearer”) is opaque and required to be introspected at runtime using the draft standard
  - What’s returned is an enhanced JWT with a new “permissions” claim that binds scopes to named resource sets
  - These are machine-readable, scope-grained, dynamic consent directives – entitlements – that an RS must act on
The AS exposes an UMA-standardized protection API to the RS

- Resource registration endpoint
- Permission registration endpoint
- Token introspection endpoint

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

- **RPT endpoint**

The **AAT** (OAuth token) protects the API and binds the RqP, client, and AS.

The client may be told: “need_info”, necessitating trust elevation for authentication or CBAC (or, through extension, ABAC).
These are *embedded* OAuth flows to protect UMA-standard security APIs

- The “PAT” and “AAT” are our names for plain old OAuth tokens – representing important UMA concepts!
  - Alice’s consent to federate authorization
  - Bob’s consent to share claims to win access
- Many “binding obligations” will hinge on their issuance
The significance of resource set registration

- The AS is authoritative for Alice’s policy
- But the RS is authoritative for what its API can do – its “verbs” and “objects”, and what Alice has created there
- Resource set registration allows the RS to remain authoritative in this fashion, and allows RS:AS to be an n:m relationship
The AS can elevate requesting party trust 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.

If the AAT was minted with too-weak authentication, the AS can request step-up for it as well.

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.
The significance of trust elevation and claims gathering

- Informing a requesting party that a resource is available for them to attempt to access (e.g. through email) is not a “magic entitlement”
- This area of the UMA protocol has variability and requires profiling and ecosystem agreement
- True “upstream” step-up authentication is possible, ensuring the entire chain is high-assurance
Authorization services architecture for high-scale environments

**Application (Requesting client)**
- Protect requested Scope
- entitlements returned
- Tokens

**Cloud/On-prem Authorization Server**
- Scopes
- Consents
- Authorization Server

**Resource Server**
- Attributes
- Entitlements
- Resource Server

**Policy Administration**
- (Scope/Consent Administration Point)

**Resource Administration**
- RS is authoritative for protected objects and scopes (verbs); AS maps to subjects

**Requesting party**
- "Consent" to release

**Resource Owner**
- "Virtualized" resource owner options

**Local entitlement enforcement**
- Requesting party
- Manage

**Decision point hands out entitlements...**
- ...directly to client apps

**Registration**
Generic simplified high-level UMA flow (spec refs: UMA = http://tinyurl.com/umacore, RSR = http://tinyurl.com/oauthsr)

UMA2. Choose resources to protect (out of band)
- UMA2. Set policies or accept system defaults (out of band)
  - RSR2. Register resource sets and scopes (ongoing)
    - UMA3.1. Attempt resource access with no token
      - UMA3.2. Register permission (resource set and scopes) sufficient for attempt
      - UMA3.2. Return permission ticket
    - UMA3.1. Return 403 with AS location and permission ticket
    - Assume any needed claims accompany request
      - UMA3.4. Request authz data with permission ticket
        - UMA3.4. Assess request against policies
          - Return RPT with success
    - Assume default RPT profile and happy path
      - UMA3.3. Introspect RPT
        - UMA3.3. Return token status and permissions
        - UMA3.1.2. Assess access attempt against permissions
          - UMA3.1.2. Return 20x with resource representation
UMA enables business logic centralization, even for “classic” access management

**Business SaaS SSO today:**
- Company X contracts with SaaS1.com
- Employees SSO in from web or native app, passing in role/group attributes
- Company X’s policies at SaaS1 govern what features users can access
- Company Y does the same at SaaS1, etc.
- Company X does the same at SaaS2, etc.

**Central authz tomorrow:**
- Company X runs an UMA AS
- SaaS1’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 SaaS1, SaaS2, etc. (authoritative “AzP” respected by each “AzRP”)
- Company Y keeps central policies for SaaS1, SaaS2, etc. (a different authoritative “AzP” respected by each “AzRP”)
IoT implications
Analysis against ACE use cases: many strong matches

☑ Owner grants different resource access rights to different parties
  • U1.1, U2.3, U3.2

☑ Owner grants different access rights for different resources on a device (including read, write, admin)
  • U1.3, U4.4, U5.2

☑ Owner not always present at time of access
  • U1.6, U5.5

☑ Owner grants temporary access permissions to a party
  • U1.7

☑ Owner applies verifiable context-based conditions to authorizations
  • U2.4, U4.5, U6.3

☑ Owner grants temporary access permissions to a party
  • U1.7

☑ Owner preconfigures access rights to specific data
  • U3.1, U6.3

☑ Owner adds a new device under protection
  • U4.1

☑ Owner puts a previously owned device under protection
  • U4.2

☑ Owner removes a device from protection
  • U4.3

☑ Owner preconfigures access rights to specific data
  • U3.1

☑ Owner revokes permissions
  • U4.6

☑ Owner grants access only to authentic, authorized clients
  • U7.1, U7.2
Profiling and extensibility enable efficiencies and non-HTTP bindings

- Protection API extensibility profile for AS-RS interactions
- Authorization API extensibility profile: for AS-client interactions
- Resource interface extensibility profile for RS-client interactions
  - E.g., to replace HTTP/TLS with CoAP/DTLS
- RPT profiling
  - E.g., to enable disconnected token introspection
- JSON extensibility all over the place
  - E.g., to enable experimentation and escape hatches
- Claim token format profiling
  - E.g., to enable a variety of deployment-specific trust frameworks
Thank you!

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