Using XACML Policies to Express OAuth Scope

Hal Lockhart

Oracle

June 27, 2013

Topics

- Scope Background
- Requirements
- Design Alternatives
- XACML Overview
- Proposed Approach
- Benefits
- Next Steps
- Future Research

Scope Background

- Issued as a result of Access Grant
- Intended to represent what is allowed by this Token (not actually defined by RFC 6749)
- Can be Handle or Self-contained
  - Handle is reference to data held by AS
  - Self-contained can be interpreted by RS
- I distinguish between Requested Scope and Issued Scope

Standardized Scope Requirements

- Self-contained by definition
- Agreed syntax and semantics
- Cover any domain
- Compute Scope given identity of Resource Owner & Client, plus requested Scope (at most)
- Determine if Scope allows requested access using info available to RS (not owner attributes)

Current Design Approaches

- Handle - reference to info held by AS
  - Tight coupling between AS & RS
  - AS could become performance bottleneck
  - AS required to manage info for token lifetime
- Role – small set of privilege alternatives
  - RS (or AS) knows what they mean (good or bad)
  - Inflexible
  - Role explosion & confusion

XACML Overview

- Access Control Policy language
- Well defined semantics
- Original syntax XML – JSON in progress
- Decision Request
  - Input: Attributes - Name, Value, Category
    - Categories identify the entities associated with request, e.g. Subject, Resource, Action, Environment, etc.
    - Set of Attributes values called a Situation
  - Output: Effect (Permit, Deny) Obligations, Advice

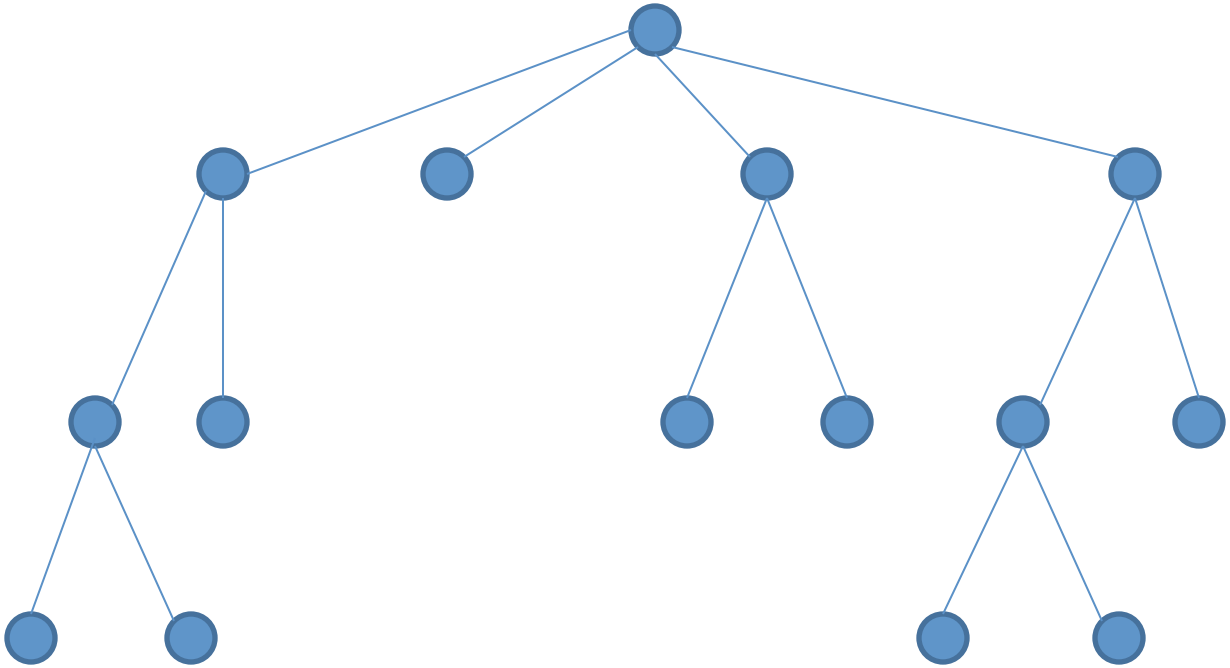
Policy Evaluation Oversimplified 1

- Policies contain operations on attribute values which produce True or False
- If False, policy is not applicable
- If True, Effect (Permit or Deny) is noted
  - (Along with Obligations and Advice)
- If multiple applicable policies have different effects, a combining rule is used
  - E.g. Default Deny

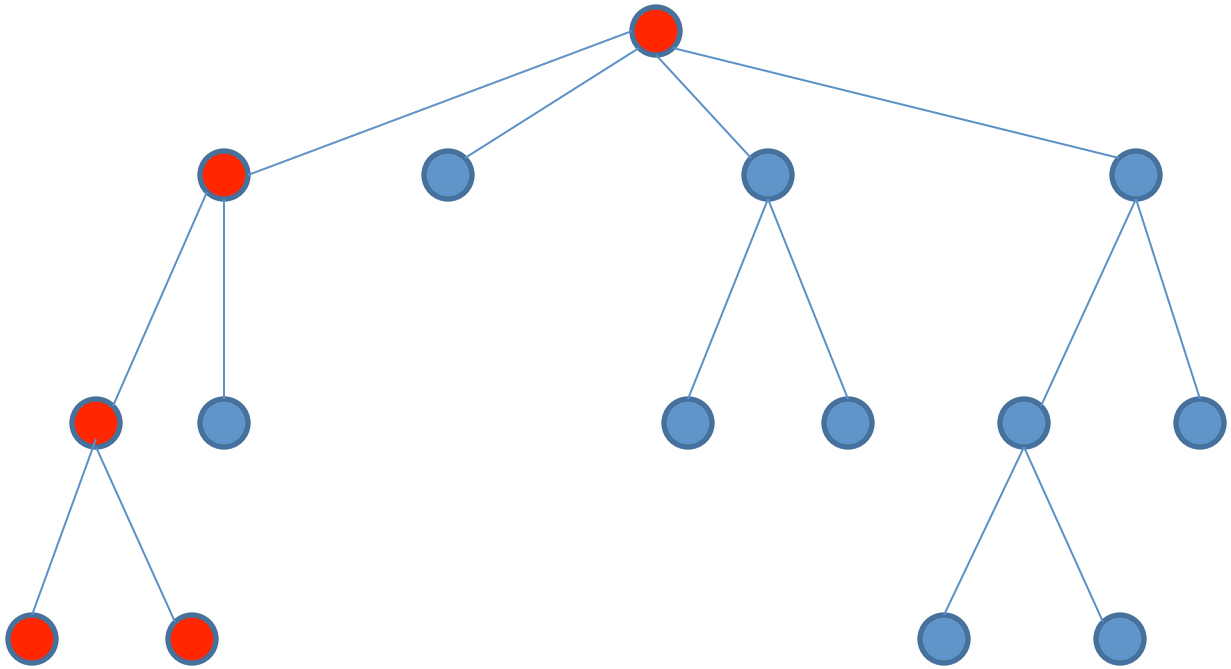
Policy Evaluation Oversimplified 2

- Policies form a tree
- Nodes are Policy Sets, leaves are Policies
- Once a branch is found not applicable, its children are not evaluated
- Policy Sets can contain Policies or Policy References
- Different policy combinations will be applicable to different Situations

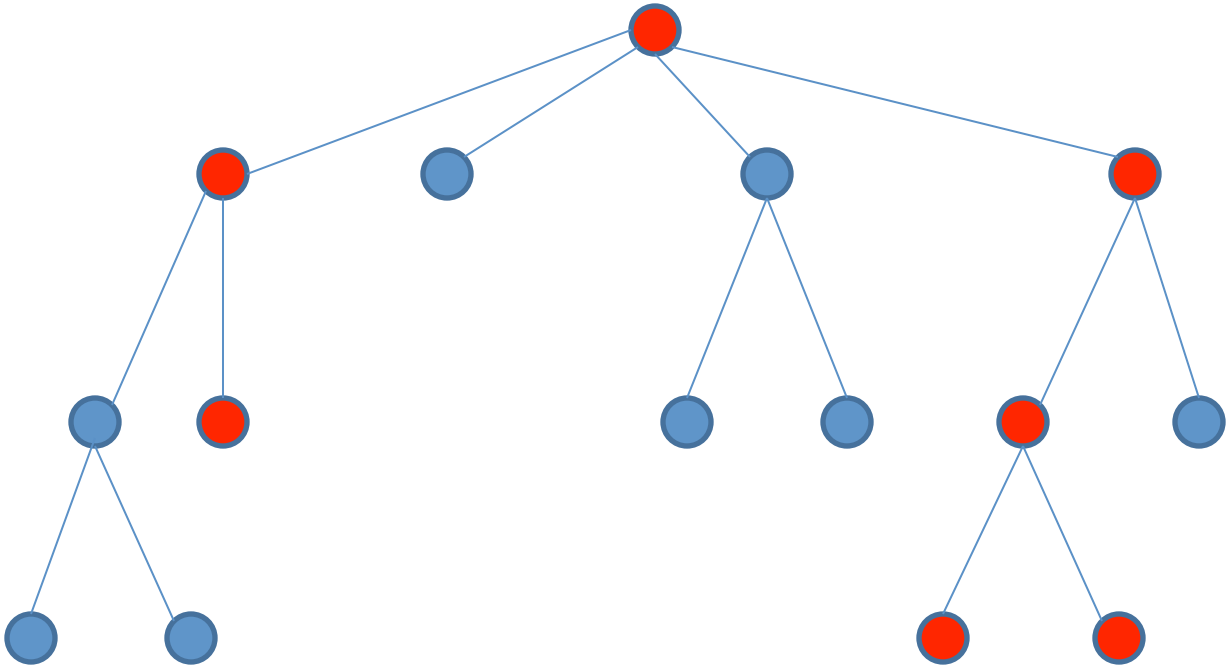
Policy Tree



Applicable Policies - Situation 1



Applicable Policies - Situation 2



Using XACML for Scope

- Scope must be able to limit
  - Resources, actions, times, dates, locations, etc.
  - Cover multiple situations
    - Non-unique attributes (Public, confidential, Tuesday)
    - Wildcards (regular expressions)
    - Boolean operators (A or B or C)
  - Could invent a new language
  - XACML already does this

Deciding What Policies

- Include all policies relevant to RS
- Write program to construct polices as needed
  - I don't know how to do this
- Small number of preset access patterns
  - Policies associated with each
  - Like Roles, except actual rights are expressed
- Use XACML PDP to select

Using XACML PDP

- Requested Scope can contain XACML decision request
  - Identify typical or most strongly protected Situation
  - Or custom tweak request to particular policies
- Submit request to PDP – 3.0 feature identify applicable policies
- Obtain policies by ID (could be cached)

Decapitated Policies

- Request contains Subject Attributes of Resource Owner
- Applicable policies most likely will reference
- Need to plug Subject Attribute values into policies as constants
- Optionally – optimize out null operations
- Eliminate dangling policy references – presumably from non-applicable policies

Decapitated Policy Example 1

Initial Policy

If Subject Attribute Group equals “user” and Resource Attribute Class equals “private” permit access.

Decapitated Policy

If “user” equals “user” and Resource Attribute Class equals “private” permit access.

Optimized Policy

If Resource Attribute Class equals “private” permit access.

Decapitated Policy Example 1

Initial Policy

If Subject Attribute Group equals “user” and Resource Attribute Name matches Regex(“/user/”+Subject Attribute Username+”/*”) permit access.

Decapitated Policy

If ”user” equals “user” and Resource Attribute Name matches Regex(“/user/”+”hal”+”/*”) permit access.

Optimized Policy

If ”user” equals “user” and Resource Attribute Name matches Regex(“/user/”+”hal”+”/*”) permit access.

or

If Resource Attribute Name matches Regex(“/user/hal/*”) permit access.

Benefits

- General, flexible, domain independent
- Means to select policies based on Client request
- Common expression of OAuth and non-OAuth access control policies
- Enables XACML delegation for full generality
- Major revisions of access model only require policy changes

Next Steps

- Increase priority of XACML JSON policies
- Need policy retrieval API standard
- Detailed specification of decapitation algorithm
- Support for just in time policies w/o delegation profile

Future Research

- Relationship to Obligations & Advice
- Relationship to other Subject Categories
- Use of XACML Admin/Delegation Profile with this scheme
- Use of access-permitted function with OAuth
- Deeper understanding of relationship between various access control models

Theoretical Considerations 1

- Use of non-unique attribute values groups entities into sub-groups and enables scaling
- Policies are written in the most natural form
  - If X, allow access
- Each policy applies to many situations
  - Also enables scaling
- As a consequence, policies only work one way
  - Determine if access allowed, can't enumerate what is allowed

Theoretical Considerations 2

- Not only can't go backward, can't go sideways
  - Can't compare two policies, only say if they are both applicable to a given situation
- Proposal is a way to sidestep problem and go from specific (request) to general (policies)
- OAuth can be looked at as a two stage policy evaluation
  1. Subject attributes for token issuance (once)
  2. Remaining attributes for access (many times)

Relationship to UMA Work

- Proposal can co-exist with anonymous Subject Attribute and Role-based scheme
- Other ideas?