Authorization

Sec PAL: A Decentralized Authorization Language

Introduction

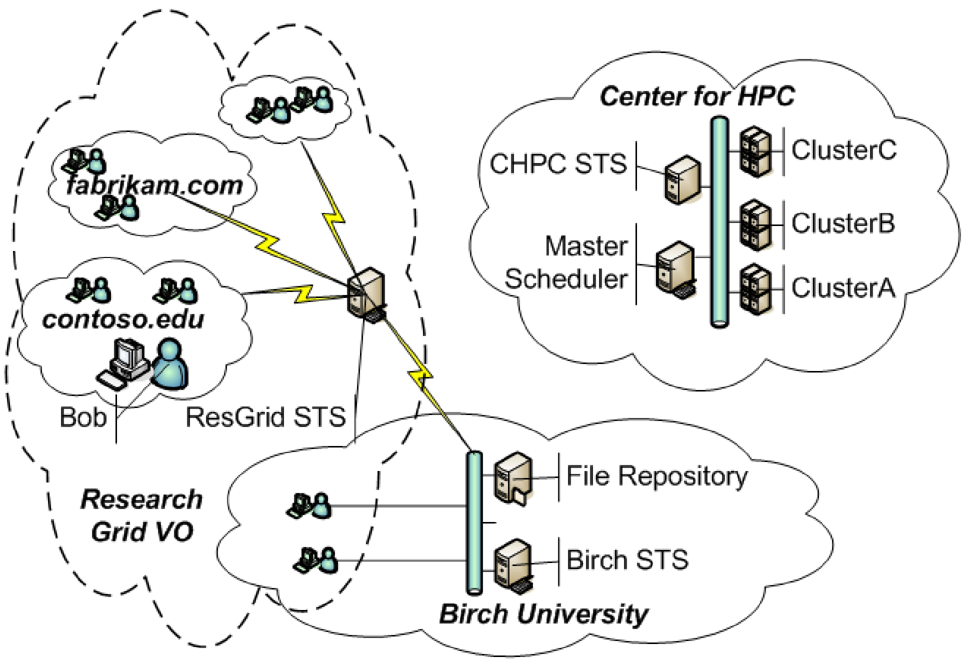
- This presentation is based on *Design and Semantics of a Decentralized Authorization Language*
- The paper describes an authorization language named SecPAL
- Agenda:
  - Problem Description
  - How SecPAL attempts to solve this problem
  - SecPAL Semantics and Syntax
  - Examples and Policy Idioms

Problem Description

- Authentication deals with the problem of how to verify identity
  - How do we know that user Alice is *really* Alice?
  - This presentation assumes that authentication is handled elsewhere
- Authorization deals with the question of what actions an identity can take
  - Is Alice allowed to read or write to a certain file?
  - Previous lectures dealt with how a single system might handle authorization

Problem Description

- Consider an ad hoc network of institutions, each with its own user base and access protocols



How SecPAL attempts to solve this problem

- Virtual organization must establish an authorization policy
- SecPAL is a declarative authorization policy language
  - Hosts asserts facts about the rights of users
  - Authorization policy consists of a collection of assertions (Assertion Context)
  - Systems can perform queries against Assertion Context
  - Provides a syntax that is easily read and reasoned by humans

SecPAL Semantics and Syntax

- An authorization policy consists of a set of assertions, called the *assertion context*
- An assertion has this form:

$$A \text{ says } fact \text{ if } fact_1, \ldots, fact_n, c$$

- *A* is the issuer, $fact_1, \ldots fact_n$ are conditional facts, and *c* is the constraint
- Conditional facts and constraints are optional
- Assertions can contain variables

Assertion Examples

- Examples:
- `STS` says `Alice` is a researcher
- `FileServer` says *x* can read `/project` if *x* is a researcher
- `Alice` says `Cluster` can read `/project` if currentTime()<=07/09/2006

Constraints

- Constraints narrow the scope of an assertion
- Can include equality, numerical inequality, path/hierarchical constraints, and regular expressions
  - FileServer says *x* can read *file* if *x* can read *dir*, *file* $\preceq$ *dir*, *x* matches *son

Delegation

- Assertions can specify who has the right to assert a fact
- Implemented using the pharse “can say”
  - `Cluster says STS can say`$_0$ x `is a researcher`
  - `Cluster has delegated the authority to assert who is a researcher to STS`
- “say_0” means that STS cannot re-delegate; “say_∞” would allow STS to re-delegate
- A fact that uses “can say” is considered *nested*, and is considered *flat* otherwise

Deduction Rules

- SecPAL provides 3 deduction rules
- Allows conclusions to be made from assertions in the assertion context

Deduction Rule “condition”

$$\text{(cond)}\ \frac{\begin{array}{c}(A \text{ says } fact \text{ if } fact_1, \ldots, fact_k \text{ where } c) \in \mathcal{AC} \\ \mathcal{AC}, D \models A \text{ says } fact_i\theta \ \text{ for all } i \in \{1..k\} \\ \models c\theta \qquad\qquad vars(fact\theta) = \emptyset\end{array}}{\mathcal{AC}, D \models A \text{ says } fact\theta}$$

- Very simply, the condition rule says that if all of the facts within an assertion are true, the entire assertion is true.

Rule “condition” Example

- Given these assertions:
  - `Cluster says` *x* `can execute dbgrep if` *x* `is a researcher`
  - `Cluster says Alice is a researcher`
- You can deduce:
  - `Alice can execute dbgrep`

Deduction Rule "can say"

$$\text{(can say)}\ \frac{\mathcal{AC}, \infty \models A \text{ says } B \text{ can say}_D\ \mathit{fact} \qquad \mathcal{AC}, D \models B \text{ says } \mathit{fact}}{\mathcal{AC}, \infty \models A \text{ says } \mathit{fact}}$$

- If *A* says that *B* can say *fact*, and *B* says *fact*, then you can deduce that *A* has asserted *fact*.

Rule "can say" Example

- Given these assertions:
  - `Cluster says STS can say` *x* is a researcher
  - `STS says` *x* is a researcher
- You can deduce:
  - `Cluster says Alice is a researcher`

Deduction Rule "can act as"

$$\text{(can act as)}\ \frac{\mathcal{AC}, D \models A \text{ says } B \text{ can act as } C \qquad \mathcal{AC}, D \models A \text{ says } C\ \mathit{verbphrase}}{\mathcal{AC}, D \models A \text{ says } B\ \mathit{verbphrase}}$$

- Asserts that all facts applicable to *C* also apply to *B*, when it is derivable that *B* can act as *C*

Rule “can act as” Example

- Given these assertions:
  - `FileServer says Node23 can act as Cluster`
  - `FileServer says Cluster can say` *x* `is a researcher`
- You can deduce:
  - `FileServer says Node23 can say` *x* `is a researcher`

Authorization Queries

- Have form *A* says *fact* and *constraints*
- Performed against a specific assertion context
- Returns an answer set of all substitutions that make the query true.
  - If the query has no variables, either an empty set or singleton set (for yes or true) is returned.

Authorization Query Example

- Assertion Context:
  - Alice says Bob can read Foo
  - Alice says Charlie can read Foo
  - Alice says David can read Foo
  - Alice says Edward can read Bar
- Authorization query:
  - Alice says *x* can read Foo
- Returns:
  - Bob, Charlie, David
  - These are all the assignments for *x* that can read Foo according to Alice

Authorization Query Table

- Contains authorization queries for a local assertion context
- Allows for parameterization of queries
  - When called, parameter is passed to the query
  - This allows an instantiated authorization query to be run against the assertion context
- Example:

`check-access-permissions(x):`

`FileServer` says *x* has access from t_1 till t_2,

$t_1 \leq$ currentTime() $\leq t_2$,

not $\exists t_3, t_4($ `FileServer` says *x* has no access from t_3 till t_4, $t_3 \leq$ currentTime() $\leq t_4)$

Authorization Query Table Example

- If called for user Alice, the query becomes:

check-access-permission(Alice) :
`FileServer` says `Alice` has access from t_1 till t_2,
$t_1 \leq$ currentTime() $\leq t_2$,
not $\exists t_3, t_4$(`FileServer` says `Alice` has no access from t_3 till t_4, $t_3 \leq$ currentTime() $\leq t_4$)

Policy Idioms

- SecPAL can be used to model a variety of authorization protocols
- Roles:
  - `NHS says FoundationTrainee can read /docs`
  - `NHS says SpecialistTrainee can act as FoundationTrainee`
  - `NHS says SeniorMedPractitioner can act as SpecialistTrainee`
  - `NHS says Alice can act as SeniorMedPractitioner`

Roles

- Roles:
  - `NHS says FoundationTrainee can read /docs`
  - `NHS says SpecialistTrainee can act as FoundationTrainee`
  - `NHS says SeniorMedPractitioner can act as SpecialistTrainee`
  - `NHS says Alice can act as SeniorMedPractitioner`
- Alice has the role of SeniorMedPractitioner, and inherits the capabilities of the SpecialistTrainee and FoundationTrainee

Bell-LaPadula

- *-Property:
  - `FileServer` says *x* can read *f* if *x* is a user, *f* is a file, level(x) >= level(*f*)
  - `FileServer` says *x* can write *f* if *x* is a user, *f* is a file, level(*x*) <= level(*f*)

- FileServer asserts that a user can read any file with a level that is the same or less than that of the user, and write to any file that has a level that is the same or greater than that of the user.

Decidability

- To be useful, authorization queries must return in a reasonable amount of time.
- The validity of a queries must be determined in a finite number of steps.
  - That is, a query must be *decidable*
- SecPAL provides definitions of safety conditions to determine whether an assertion or query is decidable.

Assertion Safety

- Assertion *A* says *fact* if $fact_1$, ..., $fact_n$,*c* is safe if and only if:
  - all conditional facts are flat
  - All variables in *c* also occur somewhere else in the assertion
  - *fact* is flat
  - All variables in *fact* occur in a conditional fact

Authorization Query Safety

- An authorization query is safe if and only if all variables in q are instantiated when query is evaluate.
- Safe: *x* says *y* can read *f*, not(*y* says *x* can read *f)*
  - All variables in the negation are instantiated by the left-hand side of the query
- Not safe: *x* says *y* can read *f*, not(*y* says *z* can read *f)*
  - *z* will not be instantiated when negation clause is evaluated

Questions?