Advanced Access Control

In many cases, identity is a *bad criteria* for authorization.

We examine two modern paradigms for access control, which overcome this limitation:

1. Role-Based Access Control
2. Trust Management Systems

Role-Based Access Control

- In many cases, authorization should be based on the function (role) of the subject in the manipulation of the object
- Examples
  - Function in a bank branch
    - teller clerks
    - financial advisors
    - Bank manager
  - Function in a hospital
    - Doctors (GP, consultant, treating doctor, …)
    - Nurses (ward nurse, nurse, …)
    - Hospital administrators
  - Functions at a university
    - Academics (teachers, research fellows, …)
    - Non-academic staff (secretaries, system administrators, …)
    - Students

Common Concepts

Definitions:

- **Active role:**
  AR(*s : subject*) = (the active role for subject *s*)
- **Authorized roles:**
  RA(*s : subject*) = {authorized roles for subject *s*}
- **Authorized transactions:**
  TA(*r : role*) = {authorized transactions for role *r*}
- **Predicate exec:**
  exec(*s : subject, t : transaction*) = true iff *s* can execute *t*
- **Session:**
  Binds a user to a set of currently activated roles

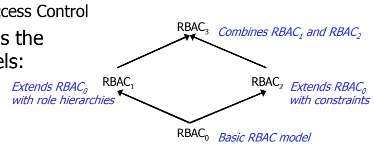
General RBAC Rules

Rules:

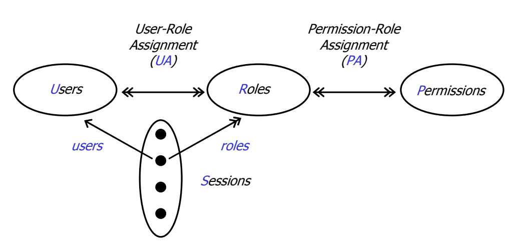
1. **Role assignment:**
   $\forall s : subject, t : transaction$ (exec(s,t) $\Rightarrow$ AR(s) $\neq \emptyset$)
   A subject can only execute a transaction if it has selected a role
2. **Role authorization:**
   $\forall s : subject$ (AR(s) $\subseteq$ RA(s))
   A subject's active role must be authorized for the subject
3. **Transaction authorization:**
   $\forall s : subject, t : transaction$ (exec(s,t) $\Rightarrow$ t $\in$ TA(AR(s)))
   A subject can only execute a transaction if it is authorized for its active role

RBAC96

- Role-Based Access Control was defined by Ferraiolo & Kuhn from NIST in 1992
- A family of related RBAC models were defined by Sandhu et al. in 1996 – this family is commonly known as RBAC96
  - RBAC96 forms the basis for most of the continued work on Role-based Access Control
- RBAC96 defines the following models:



$RBAC_0$



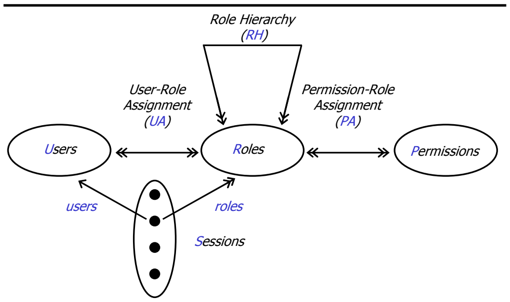
Users, Roles & Permissions

- In RBAC, a user is a human being, though it can be generalized to include other active agents
- Each individual should be known as exactly one user
- Permissions are always positive
  - No negative permission.
  - Denial of access is modelled as a constraint ($RBAC_2$ and $RBAC_3$)
- A permission can be defined for a single object or a set of objects
  - Depends on applications

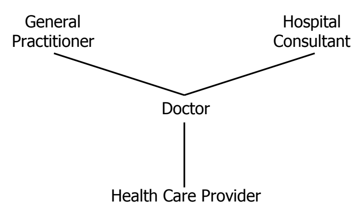
$RBAC_0$ Formal Definition

- *U, R, P,* and *S*
  - *Users, roles, permissions, sessions*
- $PA \subseteq P \times R$*: permission-role assignment*
- $UA \subseteq U \times R$*: user-role assignment*
- *user:* $S \rightarrow U$,
- *roles:* $S \rightarrow 2^R$,
  - $roles(s) \subseteq \{r \mid (user(s), r) \in UA\}$,
- *Permissions:* $S \rightarrow 2P$
  - $Permissions(s) \subseteq \cup_{r \in roles(s)}\{p \mid (p,r) \in PA\}$

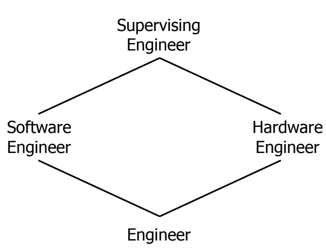
$RBAC_1$



Role Hierarchy
Transitive Inheritance of Permissions



Role Hierarchy
Multiple Inheritance of Permissions



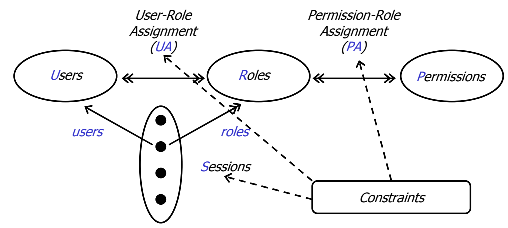
Semantics of Role Hierarchies

- User inheritance
  - r1≥r2 means every user that is a member of r1 is also a member of r2
- Permission inheritance
  - r1≥r2 means every permission that is authorized for r2 is also authorized r1
- Activation inheritance
  - r1≥r2 means that activating r1 will also activate r2

RBAC$_1$ Formal Definition

- *U, R, P, S, PA, UA* and *user* are same as RBAC$_0$
- $RH \subseteq R \times R$, a partial order with dominance relation $\geq$,
- $roles: S \rightarrow 2^R$,
  - $roles(s) \subseteq \{r \mid (\exists\, r' \geq r)\,[(user(s), r') \in UA]\}$,
- $Permissions: S \rightarrow 2^P$
  - $Permissions(s) \subseteq \bigcup_{r \in roles(s)} \{p \mid (\exists\, r'' \leq r)[(p, r'') \in PA]\}$

RBAC$_2$



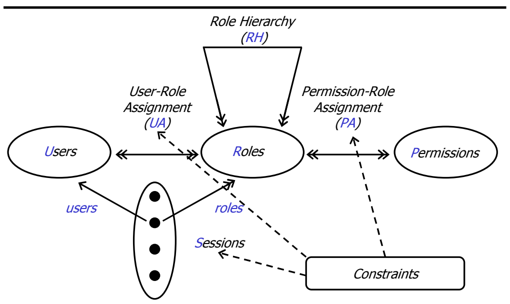
RBAC Constraints

- Mutually Exclusive Users (Separation of Duty - SoD)
  - Static Exclusion (static SoD): The same individual user can never hold mutually exclusive roles (by UA)
  - Dynamic Exclusion (dynamic SoD): The same individual user can never hold mutually exclusive roles in single session
- Mutually Exclusive Roles
  - Static Exclusion (static SoD): Two mutually exclusive roles cannot be assigned with the same permissions
  - Dynamic Exclusion (dynamic SoD): Two mutually exclusive roles can be assigned with the same permissions but cannot be activated at the same time by different users
- Mutually Exclusive Permissions
  - Static Exclusion (static SoD): The same role should never be assigned to mutually exclusive permissions
  - Dynamic Exclusion (dynamic SoD): The same role can never hold mutually exclusive permissions in single session

Dynamic Constraints

- Constraints may also consider other elements:
  - Physical environment
    - Location (certain roles can only be activated in certain places)
    - Time (time-lock on till or safe)
  - Execution history
    - Enforces well-formed transactions
  - Context
    - Business meetings
    - Emergencies
    - Games

RBAC$_3$



RBAC Summary

- Roles add a useful level of indirection, which allows aggregation of users and permissions
  - Fewer relationships to manage
    - from O(mn) to O(m+n), where m is the number of users and n is the number of permissions
- Role hierarchies allow users to assume more specialized roles (with more permissions) as needed
  - This helps enforce the principle of least privilege
- Constraints allow enforcement of separation of duty and dynamic adaptation of policies to the current context, e.g., Chinese Wall policy

Pause

Back in 15 minutes

Trust Management

- Term coined by Matt Blaze in 1996
- Provides (partial) answer to questions like:
  - "Should I perform this (dangerous) action?"
  - "Why should this principal be granted this privilege?"
- Systematic approach to managing:
  - Security policies, credentials and trust relationships
  - Based on compliance checking, not human notion of trust

Compliance Checking

- Provides advice to applications on whether "dangerous" actions should be permitted
- Compliance checker uses local policy and signed credentials in these decisions
  - Only actions that conform to policy are allowed
- As long as all dangerous actions are checked with the compliance checker, we know that the security policy is being followed

Distributed Policies

- Ideally policies are stored in one place and specified by one person
- In reality, different parts of the policy often come from different places (and authorities)
  - Delegation of authorization
  - Different administrators for different services
  - Multiple requirements for access
- There may not even be a single complete statement of the policy
- Large scale systems imply high complexity in managing specification, location and consistency of policy components

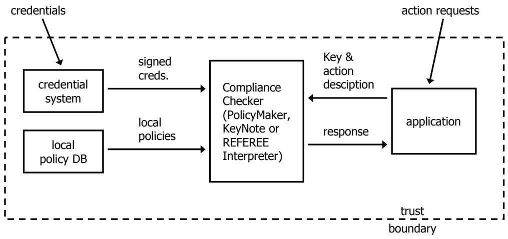
Policies and Credentials

- A policy specifies *who* is allowed to do *what*
  - *who* may be a public-key
  - *what* may be a potentially dangerous action
- A credential delegates authorization to *someone else*
  - *someone else* may also be a public-key
- Distributed systems blur the distinction between policies and credentials
  - A credential is a policy signed by someone who are authorized to do so

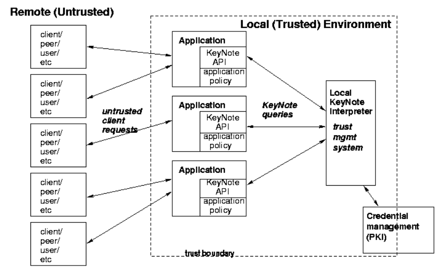
Trust Management Elements

- A language for *Actions*
  - Operations with security implications for applications
- A naming scheme for *Principals*
  - Entities that can be authorized to request actions
- A language for *Policies*
  - Govern the actions that principals are authorized for
- A language for *Credentials*
  - Allow principals to delegate authorization
- A *Compliance Checker* and interface
  - Service that determines whether a requested action should be allowed, based on policy and a set of credentials

Trust Management Architecture



KeyNote



Typical Trust Management Languages

- PolicyMaker
  - Blaze, Feigenbaum and Lazy (1996)
  - Compliance checking formalized in Blaze, Feigebaum and Strauss (1998)
  - Very general, designed more for study than for use
- KeyNote
  - Blaze, Feigenbaum, Ioannidis, Keromytis (1997)
  - Defined in RFC 2704
  - Designed to be used, especially in Internet apps.
- Both share the same basic semantic structure
  - Based on assertions

Assertions

- Authorize principals to perform actions
- Policies are defined by a collection of assertions
- Assertions contain two basic parts:
  - A principal identifier (key or key expression)
    - allows the principal to be authenticated
  - Action predicate
  - Principal is authorized to perform actions that pass the action predicated
- Assertions can be signed by keys
  - Signed assertions are credentials

Assertion Syntax

- *principal* **is-authorized-for** *predicate* {**signed-by** *authorizer*}
  - The keys in *principal* are authorized for actions that pass the *predicate* according to *authorizer*
- Exmple of Keynote assertion syntax:

  Authorizer: <keyword POLICY or signer's public-key>
  Licensees: <principal, tests signer keys>
  Conditions: <trust-expression, tests action attributes>
  Signature: <encoding of signature, for signed credentials>

Compliance Checking Semnatics

- An action is allowed if any policy assertion allows it
- An assertion is considered to allow an action if its predicate passes and either
  - The action was directly requested by the assertions licensees
  - The action was approved by some other assertion signed by the licensees

Compliance Checking Process

- Application collects appropriate assertions
  - Local, trusted root policy assertions
  - Credentials signed by someone else
- Application forms action description
  - Collection of free form attributes
  - Associated with principal identifier (ID or key)
- Compliance Checker finds compliance value
  - Evaluates action against conditions in assertions forming a graph between root policy and requestor
  - Binary: allowed/denied, or multi-valued

Assertion Monotonicity

- Assertions are monotone
  - Adding an assertion can never cause an acction to become disallowed
  - Deleting an assertion will never cause a disallowed action to become allowed
  - Nothing is allowed unless explicitly allowed by an assertion
- Implications of monotonicity
  - Safe for distributed systems
    - Missing assertions cannot cause policy violations
  - Set of allowing assertions constitute "proof of compliance"
    - Clients can collect appropriate signed assertions and present them to server
  - No conflicts
    - If an action can be allowed it will be allowed

Summary of Trust Management

- Appropriate for large-scale distributed systems
  - Decentralized policy specification and storage
  - Decentralized (autonomous) policy enforcement
- Provides both decision and reason for decision
  - List of credentials used to authorize request
- Facilitates dynamic evolution of policy
  - Incremental addition of assertions allows policies to evolve, e.g., adding new principals/roles/permissions/resources
  - Well suited for dynamic open systems (pervasive computing)