A Framework for Enforcing Constrained RBAC Policies

Jason Crampton       Hemanth Kambhammettu
Information Security Group
Royal Holloway, University of London

SecureCom 2009
One-page overview

Constraints specify “prohibited” authorization states

Preventing constraint violation is necessary
  ▶ Why? – Enforce system security policy
  ▶ How? – Deny access requests that would cause system to enter state in which some constraint is violated

Constraints must be evaluated when deciding access requests

Decision-making process becomes more complex in the presence of constraints

**Aim:** Simplify decision-making process
RBAC

Constraints

Deciding Access Requests

Concluding Remarks

Questions
Access control

In most situations it is essential to limit access to protected resources

- Authorization policy identifies those interactions between users and resources that are authorized

Access control mechanism comprises policy enforcement point (PEP) and policy decision point (PDP)

- PEP “traps” attempted interactions by users
- PDP evaluates interactions with respect to authorization policy
- PEP enforces (authorization) decisions of PDP

Access control model identifies entities and relationships and specifies how interactions are to be evaluated
Role-based access control (RBAC) introduces the concept of a role

- Users are authorized to act in certain roles
- Roles are authorized for certain permissions (resource-action pairs, which model interactions)

More formally

- We have a set of users $U$, a set of roles $R$ and set of permissions $P$
- Permissions are assigned to roles $PA \subseteq P \times R$
- Users are assigned to roles $UA \subseteq U \times R$
Sessions and constraints

RBAC also includes the idea of a user session

- User creates a session to interact with system
- User activates a subset of assigned roles in a session
- Access decisions are made based on the roles that are active in a session

RBAC also includes constraints

- No user may be assigned to $r_1$ and $r_2$ (“static separation of duty”)
- No user may activate $r_1$ and $r_2$ in a session (“dynamic separation of duty”)

The RBAC model does not provide any guidance on constraint enforcement
RBAC state

Different state components
  - **Static** – defines RBAC policy components
    - user-role and permission-role assignments
  - **Dynamic** – defines “runtime” environment
    - user sessions, role activations and permission invocations
  - **Historic** – defines previous system activity
    - successful role activations and permission invocations

Authorization state evolves over time
  - RBAC policy components may change (as users are assigned to or revoked from roles, . . .)
  - Dynamic (and historic) state may change (as users create sessions, are granted permissions, . . .)
RBAC

Constraints

Deciding Access Requests

Concluding Remarks

Questions
Constrained RBAC policies

Different types of constraints

- **Static** – based on configuration of system
  - restrict evolution of administrative state
  - example: user $u$ cannot be assigned to both roles $r_1$ and $r_2$

- **Dynamic** – based on current state of system
  - restrict evolution of runtime state
  - example: user $u$ cannot simultaneously activate roles $r_1$ and $r_2$

- **Historic** – based on all previous states of system
  - restrict evolution of historic state
  - example: user $u$ can never invoke permission $p_2$ having invoked $p_1$ (and vice versa)
Formally, we specify a constraint as a 4-tuple \((D, T, k, c)\)

- \(D\) is the **domain** – the set to which the constraint applies
- \(T\) is the **target** – the entities for which the association of members of \(D\) is constrained
- \(t\) is the maximum number of elements to be permitted in \(\{d\} \times T\) for each \(d \in D\)
- \(c \in \{\text{static}, \text{dynamic}, \text{historic}\}\) is the context (state component) within which the constraint should be evaluated
Constraint evaluation

Constraints are evaluated by referring to (part of) authorization state

- $D$, $T$ and $c$ define the evaluation context of a constraint
- Constraint $(U, \{r_1, r_2\}, 1, s)$ requires that at most 1 role from \{\(r_1, r_2\}\} may be assigned to a user $u \in U$
  - A request to assign a user to $r_1$ or $r_2$ may result in the violation of this constraint
  - We evaluate the request by considering the $UA$ relation
- Constraint $(S, \{r_1, r_2, r_3, r_4, r_5\}, 3, d)$ requires that at most 3 roles (from \{\(r_1, \ldots, r_5\}\}) may be activated in a session $s \in S$
  - A request to activate a role may result in the violation of this constraint
  - We evaluate the request by considering the dynamic state
RBAC

Constraints

Deciding Access Requests

Concluding Remarks

Questions
Deciding access requests (1)

Decision-making (in the presence of authorization constraints) requires two checks

- Authorization checking
  - Is user authorized to perform this operation?
- Constraint checking
  - Would granting request cause violation of one or more constraints?

Observations

- Authorization checking is simple
- Constraint checking is more complex
- Decision-making becomes complex activity

We simplify the decision-making process by transforming constraint checking into authorization checking
Deciding access requests (2)

A constraint \((D, T, k, c)\) is said to be violation-prone if there exists \(d \in D\) such that \(d\) is associated with \(T' \subseteq T\) with \(|T'| = k - 1\)

- All requests that would lead to \(d\) being associated with \(t \in T \setminus T'\) should be denied
- We call such requests prohibited

Suppose we have constraint \((S, \{r_1, \ldots, r_5\}, 3, d)\) and user \(u\) has activated a session containing \(\{r_1\}\)

- A request from \(u\) to activate \(r_2\) will be granted (assuming \(u\) is authorized for \(r_2\))
- The constraint is now violation-prone
- Requests to activate \(r_3, r_4\) and \(r_5\) are all prohibited
Deciding access requests (3)

Having granted an access request

1. Identify violation-prone constraints
2. Determine all prohibited requests for those violation-prone constraints
3. Append prohibited set of requests to negative authorization policy

Access control mechanism implements a first-applicable with deny-default decision strategy

1. if negative policy prohibits request then Deny
2. else if positive policy authorizes request Permit
3. else Deny
RBAC

Constraints

Deciding Access Requests

Concluding Remarks

Questions
Advantages of our approach

Constraint checking is **not** part of the decision-making process

- Constraints are only evaluated following success of constrained request
- Reduces the number of times constraints are evaluated

Decision-making times become **lower** than existing approaches

- Constraints are enforced by querying negative authorization information
- Existing approaches evaluate constraints for deciding every request
Overheads of our approach

Maintain negative authorization state
  ▶ Space is used as trade-off for quick access decisions

Accuracy of updates to negative authorization state is crucial for constraint enforcement
  ▶ False positives
    ▶ request violates a constraint, but is granted
    ▶ cause constraint violation
  ▶ False negatives
    ▶ request does not violate a constraint, but is denied
    ▶ denial of permissable access requests
RBAC

Constraints

Deciding Access Requests

Concluding Remarks

Questions