Constraints in Role-Based Access Control

Jason Crampton

Information Security Group, Royal Holloway University of London

jason.crampton@rhul.ac.uk
www.isg.rhul.ac.uk/~jason
Outline

• Introduction
  – Separation of duty
  – Role-based access control
  – Observations and motivation

• Constraints
  – Specification
  – Interpretation and implementation

• Concluding remarks
  – Matters arising
  – Related work
Separation of duty

- Well known control principle in management
  - “Mission critical” combination of tasks required to be performed by different people
  - Prevents accidental or malicious violation of business requirements

- Examples
  - A cheque must be raised and authorised by two different people
  - A nuclear missile must be armed and launched by two different people
Varieties of separation of duty

• Static
  – No user can have the opportunity to access both $file_1$ and $file_2$
  – Based on configuration of system (access control matrix, say)

• Dynamic
  – No user can have access to both $file_1$ and $file_2$ at the same time
  – Based on current state of system (*-property in Bell-LaPadula)

• Historical
  – No user can have accessed both $file_1$ and $file_2$
  – Based on all previous states of the system (history matrix in Chinese Wall model)
  – Important in workflow systems (Bertino et al)
Role-based access control

- Introduces the concept of a role, a permission and a role hierarchy
  - Permissions assigned to roles \((PA \subseteq P \times R)\)
  - Users assigned to roles \((UA \subseteq U \times R)\)
- A permission is an association between an action (read, write, etc.) and an object (Simon & Zurko, 1997)
  - A permission can be thought as an object-method pair or a class-method pair in an object-oriented environment
- Role hierarchy
  - Aggregates permissions and implicitly assigns users to roles
    - Simplifies administration
  - Models hierarchical organisational structure of enterprise
    - Can be used to identify access control requirements (including separation of duty)
Some immediate observations

- Separation of duty procedures are implemented to prevent some specific “bad” combination of things occurring
- Separation of duty raises two issues
  - Specification (What do we want to prevent?)
  - Enforcement (How do we prevent it?)
- Specification is distinct from enforcement
- Specification is not complicated!
  - Identify the bad combination
- A specification scheme must admit the articulation of historical constraints
  - Static separation of duty is not a practical or realistic variation of separation of duty … it does not capture most real-world organisational control principles (Simon & Zurko, 1997)
- Enforcement of historical constraints may be hard
  - How do we keep track of previous events?
Specification

- A separation of duty constraint is specified as a triple
  - \((\text{constraint\_scope}, \text{constraint\_set}, \text{temporal\_context})\)
    - \(\text{constraint\_scope}\) specifies the set to which the constraint applies
    - \(\text{constraint\_set}\) specifies the entities (that form a bad combination with respect to the scope)
    - \(\text{temporal\_context} \in \{\text{static, dynamic, historical}\}\)

- \((U, \{\text{fin\_clerk, po\_clerk}\}, \text{static})\)
  - “no user can be assigned to both the finance clerk and purchasing clerk roles”
  - \(U\) is the scope of the constraint
Specification – further examples

- \((R, \{\text{po.approve, cheque.raise}\}, \text{static})\)
  - “no role can (have the permission to) approve purchase orders and raise cheques”

- \((\text{jason, \{fin_clerk\}}, \text{static})\)
  - “jason cannot be assigned to the finance clerk role”

- \((U, \{\text{cheque.raise, cheque.issue}\}, \text{historical})\)
  - “no user can raise and issue the same cheque”

- \((U, \{A\text{objects, Bobjects}\}, \text{historical})\)
  - “no user can access the objects owned by \(A\) having accessed-objects owned by \(B\) and vice versa” (Chinese Wall-ish model)
Interpretation and enforcement

- \((U, \{\text{fin}_\text{clerk}, \text{po}_\text{clerk}\}, \text{static})\) is satisfied if for all users \(u\), \(\{(u, \text{fin}_\text{clerk}), (u, \text{po}_\text{clerk})\}\) is not a subset of \(UA\)
  - Static constraints can be enforced by examining the appropriate component of the model (\(RH, UA\) or \(PA\))

- \((U, \{\text{cheque}.\text{raise}, \text{cheque}.\text{issue}\}, \text{historical})\) is satisfied if for every instance of a \(\text{cheque}\) object, no user has invoked both the \(\text{raise}\) and \(\text{issue}\) method on that object
  - It is more difficult to enforce a historical constraint
Enforcement of historical constraints

Existing methods

- Maintain a record of all permissions that have been invoked
  - The Chinese Wall model for object-based separation of duty implemented this approach using a history matrix
- Compute all possible valid requests given execution history and only subsequently allow such requests
  - This approach has been used in workflow environments

Proposed method

- Create blacklist of requests that would cause a constraint to be violated
  - Implemented using request “blacklists”
  - Blacklists are not required if there are no historical constraints
Blacklists

- User-role constraint generates a list of prohibited user-role assignments
- Permission-role constraint generates a list of prohibited permission-role assignments
- User-permission or role-permission constraint generates a list containing prohibited permissions
  - If \((U, \{p_1, p_2\}, \text{historical})\) is a constraint and \(jason\) invokes permission \(p_1\) then \((jason, p_2)\) is a prohibited permission invocation
  - Can be implemented as an “anti-role” assigned to \(jason\)
Example

- **Constraints**
  - \((U, \{r_1, r_2\}, \text{static})\)
  - \((\text{jason}, \{r_1\}, \text{static})\)
  - \((U, \{p_1, p_2\}, \text{historical})\)

- **Initial configuration**
  - \(UA = \emptyset\)
  - \(L_{UA} = \{(\text{jason}, r_1)\}\)

- **AssignUser(mick, r_1)**
  - \(UA := \{(\text{mick}, r_1)\}\)
  - \(L_{UA} := \{(\text{jason}, r_1), (\text{mick}, r_2)\}\)

- **UsePerm(mick, p_1)**
  - \(L_{\text{mick}} := \{p_2\}\)

- **RevokeUser(mick, r_1)**
  - \(L_{UA} := \{(\text{jason}, r_1)\}\)
    - Updated – static constraint applies
    - \(L_{\text{mick}} = \{p_2\}\)
    - Unchanged – historical constraint applies
Matters arising

- It is not possible to have more than two elements in a constraint set
  - We cannot specify a constraint of the form “Any user can invoke two of three methods on an object”
  - The best we can do is to allow a user to invoke at most one method from two
  - Discussed in detail in paper
- The blacklists may prevent some request from succeeding indefinitely
  - A poorly specified set of constraints may lead to the situation where no user can invoke a particular method on a particular object
  - Can we offer any guarantees?
- Implementation of the model as a generic application-independent tool for specifying and implementing constraints
  - Extend the security classes in Java 2?
- Section 3.2.1: first bullet should read
  - “… assigned to both $u_1$ and $u_2$” not
  - “… assigned to both $r_1$ and $r_2$”
## Related work

### Specification
- Simon & Zurko (1997)
  - Rule-based
- Gligor *et al.* (1998)
  - First order logic
  - Transformation into fragment of first order logic
- Jaeger & Tidswell (2001)
  - Syntax-heavy set-based approach
- Bertino *et al.* (2001)
  - Rule-based constraints in workflow management systems

- All the above schemes include a specification of
  - the components of the constraint and
  - the conditions that must obtain for the constraint to be satisfied

### Enforcement
- Transaction control expressions (Sandhu 1988)
  - *Transient objects* are annotated with the identity of the user that invoked the methods on an object
- Adage implementation (Simon & Zurko 1997)
  - History matrix (generalized from Chinese Wall model)
- Edjlali *et al.* (1998)
  - The access rights available to mobile code are controlled in a dynamic fashion using *events* and *handlers*
- Bertino *et al.* (2001)
  - All possible valid execution paths are computed (using stratified logic programming techniques)
  - An access request is granted if it is the next step in a valid execution path
- Fournet & Abadi (2003)
  - Access rights available to mobile code are determined by execution history