

Role-Based Access Control (RBAC)

Scott D. Stoller

CSE592: Security Policy Frameworks



References

- Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. *Role-Based Access Control Models*. *IEEE Computer* 29(2):38-47, February 1996.
- Role-Based Access Control. ANSI INCITS Standard 359-2004, February 2004.
 - ◆ Defines an RBAC model: abstract data structures, API
 - ◆ Goal: Help consumers who struggled with different variants of RBAC in different DBMSs, etc.
 - ◆ SQL:1999 also defines an RBAC model
 - ◆ Doesn't consider policy administration

Roles

- **Role**: an abstraction associated with a set of permissions, typically associated with a function or position in an organization.
 - ◆ **Examples**: doctor, nurse, patient, receptionist
- RBAC policy specifies:
 - ◆ the roles that each user may adopt
 - ◆ the permissions associated with each role.
 - Permission = [operation, resource]
 - Operation may be resource-specific, not limited to read/write.

User-Role and Permission-Role Assignments



- A user has a permission if he is a member of some role with that permission.
- **Benefits of RBAC**
 - ◆ Greatly reduces redundancy: there is a set of permissions for each role, not each user
 - ◆ Easy to administer: A few role assignments are sufficient to handle a new user, job change, etc.
 - Roles reflect organizational structure, so changes to PR are relatively rare. "Role mining"

Roles vs. Groups: Quotes from [Samarati+ 2001]

- "groups define sets of users while roles define sets of privileges."
 - ◆ A role can be regarded as a set of **users** (via UR) and a set of **permissions** (via PR).
 - ◆ Similar for (e.g.) UNIX groups. But finding all permissions of a group might be inefficient.
- "roles can be "activated" and "deactivated" by users at their discretion, while group membership always applies, that is, users cannot enable and disable group memberships (and corresponding authorizations) at their will."
 - ◆ In other words, roles are groups with additional features: activation, perhaps hierarchy, SoD, etc.

Roles vs. Groups

- **Traditionally**, roles and groups are administered differently
 - ◆ **Groups**: **centralized** administration of **membership**, **ownership**-based administration of permissions.
 - cf. previous point about finding all perms of a group
 - ◆ **Roles**: **centralized or hierarchical** administration of **membership and permissions** (role administrator knows what permissions are needed for the job functions).
 - All objects implicitly "owned" by the organization.
 - Recent admin models for RBAC [Li and Mao 2007] are more flexible and can consider ownership.
 - Admin in SQL: ownership-based and decentralized

Role Activation

- A user must **activate** a role in a **session** in order to use the permissions associated with that role.
- **Example:** Dr. Smith is sometimes a doctor, sometimes a patient.
- **Analogy:** system administrator sometimes logs in as root, sometimes as a regular user
- Helps enforce the **principle of least privilege**.
- **Limits potential damage** due to human errors, software defects, etc.
- A **session** (a window, application, ...) belongs to one user. Multiple roles may be active in it.

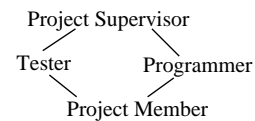
Sept. 2007

Scott Stoller, Stony Brook University

7

Role Hierarchy

- $r1 \supseteq r2$ (**r1 inherits from r2**, **r1 is senior to r2**) means every member of r1 is also a member of r2. thus, members of r1 have all the permissions that members of r2 have.
- Permission flows **up**. Membership flows **down**.
- RH further reduces redundancy and eases administration.
 - ◆ New supervisor is added to one role, instead of four.
 - ◆ New perm for project members is added to 1 role, not 4.



Sept. 2007

Scott Stoller, Stony Brook University

8

Static Separation of Duty (SSoD)

- **[Simplified]** An **SSoD constraint** is a pair of roles whose membership must be disjoint, i.e., every user is in at most one of them.
 - ◆ **Example:** (PurchasingClerk, ReceivingClerk), (PurchasingClerk, AccountingClerk), (ReceivingClerk, AccountingClerk)
- An **SSoD constraint** is a set S of roles and a threshold n. Every user may be in at most n roles in S.
 - ◆ **Example:** ({PurchasingClerk, AccountingClerk, ReceivingClerk}, 1)
- SSoD is enforced during policy administration.

Sept. 2007

Scott Stoller, Stony Brook University

9

Dynamic Separation of Duty (DSoD)

- Dynamic SoD constrains **role activation** instead of **role membership**.
- An **SSoD constraint** is a set S of roles and a threshold n. At most n roles in S can be activated in each session.
- **Object-based SoD** is also dynamic and is more realistic, but is not included in the ANSI Standard.
 - ◆ That would require introducing a notion of transaction (or workflow) and keeping track of history.
 - ◆ They wanted to keep the standard simple.

Sept. 2007

Scott Stoller, Stony Brook University

10

Administrative Policy

- ANSI Standard covers admin ops but not admin policy.
- **Administrative RBAC 1997 (ARBAC97)** [sandhu99arbac]
 - ◆ **administrative roles** (in addition to regular roles)
 - ◆ **administrative permissions** (for adding and removing users and permissions from regular roles)
- **Example:** **Project leader** can add/remove users/permissions from roles on the project he/she leads.
- Who can change the ARBAC policy? ARBAC doesn't say
- Details when we read [Li and Mao, 2007]

Sept. 2007

Scott Stoller, Stony Brook University

11

Formal Model of RBAC

- The ANSI Standard gives a formal model in Z, an ANSI standard language based on logic and set theory.
- We wrote a simplified and easier-to-read model in a simple functional programming language.
- Yanhong A. Liu and Scott D. Stoller. Role-Based Access Control: A Corrected and Simplified Specification. In Cliff Wang, et al., editors, *DoD Sponsored Information Security Research: New Methods for Protecting Against Cyber Threats*. Wiley, 2007.

Sept. 2007

Scott Stoller, Stony Brook University

12

Model of Core RBAC

```
class CoreRBAC:
... // reference model
... // functionalities

OBJS: set(Object)
OPS: set(Operation)
USERS: set(User)
ROLES: set(Role)
PR: set(tuple(tuple(Operation, Object), Role))
UR: set(tuple(User, Role))
SESSIONS: set(Session)
SU: set(tuple(Session, User))
SR: set(tuple(Session, Role))
```

Sept. 2007

Scott Stoller, Stony Brook University

13

Administrative Commands

```
AddUser(user):
pre-condition: user notin USERS;
USERS = USERS + {user}

AddRole(role):
pre-condition: role notin ROLES;
ROLES = ROLES + {role}

AssignUser(user,role):
pre-condition: user in USERS, role in ROLES, [user,role] notin UR;
UR = UR + {[user, role]}
```

Sept. 2007

Scott Stoller, Stony Brook University

14

Administrative Commands

```
GrantPermission(operation, object, role):
pre-condition: operation in OPS, object in OBJS, role in ROLES,
[[operation,object],role] notin PR;
PR = PR + {[operation,object],role}

DeleteUser(user):
pre-condition: user in USERS;
UR = UR - {[user,r]: r in ROLES}
for s in SESSIONS | [s,user] in SU:
DeleteSession(user,s)
USERS = USERS - {user}
```

Sept. 2007

Scott Stoller, Stony Brook University

15

Administrative Commands

```
DeleteRole(role):
pre-condition: role in ROLES;
PR = PR - {[op,obj],role}: op in OPS, obj in OBJS
UR = UR - {[u,role]: u in USERS}
for s in SESSIONS, u in USERS | [s,u] in SU, [s,role] in SR:
DeleteSession(u,s)
ROLES = ROLES - {role}

DeassignUser(user, role):
pre-condition: user in USERS, role in ROLES, [user,role] in UR;
UR = UR - {[user,role]}
for s in SESSIONS | [s,user] in SU, [s,role] in SR:
DeleteSession(user,s)
```

Sept. 2007

Scott Stoller, Stony Brook University

16

Administrative Commands

```
RevokePermission(operation, object, role):
pre-condition: operation in OPS, object in OBJS, role in ROLES,
[[operation,object],role] in PR;
PR = PR - {[operation,object],role}
```

Sept. 2007

Scott Stoller, Stony Brook University

17

Supporting System Functions

```
CreateSession(user, session, ars):
pre-condition: user in USERS, session notin SESSIONS,
ars subset AssignedRoles(user);
SU = SU + {[session,user]}
SR = SR + {[session,r]: r in ars}
SESSIONS = SESSIONS + {session}

DeleteSession(user, session):
pre-condition: user in USERS, session in SESSIONS,
[session,user] in SU;
SU = SU - {[session,user]}
SR = SR - {[session,r]: r in ROLES} // maintain SR
SESSIONS = SESSIONS - {session}
```

Sept. 2007

Scott Stoller, Stony Brook University

18

Supporting System Functions

AddActiveRole(user, session, role):
pre-condition: user in USERS, session in SESSIONS, role in ROLES,
[session,user] in SU, [session,role] notin SR,
role in AssignedRoles(user);
SR = SR + {[session,role]}

DropActiveRole(user, session, role):
pre-condition: user in USERS, session in SESSIONS, role in ROLES,
[session,user] in SU, [session,role] in SR;
SR = SR - {[session,role]}

CheckAccess(session, operation, object):
pre-condition: session in SESSIONS, operation in OPS, object in OBJS;
return exists r in ROLES | [session,r] in SR, [[operation,object],r] in PR

Sept. 2007 Scott Stoller, Stony Brook University 19

Review Functions

AssignedUsers(role):
pre-condition: role in ROLES;
return {u: u in USERS | [u,role] in UR}

AssignedRoles(user):
pre-condition: user in USERS;
return {r: r in ROLES | [user,r] in UR}

Sept. 2007 Scott Stoller, Stony Brook University 20

Advanced Review Functions

RolePermissions(role):
pre-condition: role in ROLES;
return {[op,obj]: op in OPS, obj in OBJS | [[op,obj],role] in PR}

UserPermissions(user):
pre-condition: user in USERS;
return {[op,obj]: r in ROLES, op in OPS, obj in OBJS |
[user,r] in UR, [[op,obj],r] in PR}

SessionRoles(session):
pre-condition: session in SESSIONS;
return {r: r in ROLES | [session,r] in SR}

Sept. 2007 Scott Stoller, Stony Brook University 21

Advanced Review Functions

SessionPermissions(session):
pre-condition: session in SESSIONS;
return {[op,obj]: r in ROLES, op in OPS, obj in OBJS |
[session,r] in SR, [[op,obj],r] in PR}

RoleOperationsOnObject(role, object):
pre-condition: role in ROLES, object in OBJS;
return {op: op in OPS | [[op,object],role] in PR}

UserOperationsOnObject(user, object):
pre-condition: user in USERS, object in OBJS;
return {op: r in ROLES, op in OPS | [user,r] in UR, [[op,object],r] in PR}

Sept. 2007 Scott Stoller, Stony Brook University 22

Model of Hierarchical RBAC

INH: set(tuple(Role,Role)) // inheritance relation

INH is not necessarily reflexive or transitive.

Acyclicity Requirement

forall r1, r2 in ROLES | [r1,r2] in INH*, [r2,r1] in INH* => r1=r2

Single Inheritance Requirement (for RBAC with limited hierarchy)

forall r, r1, r2 in ROLES | [r,r1] in INH, [r,r2] in INH => r1=r2

Sept. 2007 Scott Stoller, Stony Brook University 23

Administrative Commands

AddInheritance(heir,bearer): // for unrestricted inheritance
pre-condition: heir in ROLES, bearer in ROLES,
[heir,bearer] notin INH;
INH = INH + {[heir,bearer]}

AddInheritance(heir,bearer): // for general hierarchy
... // same as above except to add, in the pre-condition,
// [bearer,hier] notin INH*, to check acyclicity

AddInheritance(heir,bearer): // for limited hierarchy
... // same as above except to also add, in the pre-condition,
// not exists r in ROLES | [heir,r] in INH, to check single inheritance

Sept. 2007 Scott Stoller, Stony Brook University 24

Administrative Commands

DeleteInheritance(heir,bearer):
pre-condition: heir in ROLES, bearer in ROLES, [heir,bearer] in INH;
INH = INH - {[heir,bearer]}

AddAscendant(heir,bearer):
AddRole(heir)
AddInheritance(heir,bearer)

AddDescendant(bearer,heir):
AddRole(bearer)
AddInheritance(heir,bearer)

Sept. 2007 Scott Stoller, Stony Brook University 25

Supporting System Functions

CreateSession(user, session, ars):
... // same as in CoreRBAC except that, in the precondition,
// AssignedRoles is replaced with AuthorizedRoles,
// which is defined below

AddActiveRole(user, session, role):
... // same change as for CreateSession above

Sept. 2007 Scott Stoller, Stony Brook University 26

Review Functions

AuthorizedUsers(role):
pre-condition: role in ROLES;
return {u: heir in ROLES, u in USERS | [heir,role] in INH*,
[u,heir] in UR}

AuthorizedRoles(user):
pre-condition: user in USERS;
return {r: heir in ROLES, r in ROLES | [user,heir] in UR,
[heir,r] in INH*}

Sept. 2007 Scott Stoller, Stony Brook University 27

Advanced Review Functions

RolePermissions(role):
pre-condition: role in ROLES;
return {[op,obj]: bearer in ROLES, op in OPS, obj in OBJS |
[role,bearer] in INH*, [[op,obj],bearer] in PR}

UserPermissions(user):
pre-condition: user in USERS;
return {[op,obj]: heir in ROLES, bearer in ROLES, op in
OPS, obj in OBJS |
[user,heir] in UR, [heir,bearer] in INH*,
[[op,obj],bearer] in PR}

Sept. 2007 Scott Stoller, Stony Brook University 28

Advanced Review Functions

RoleOperationsOnObject(role, object):
pre-condition: role in ROLES, object in OBJS;
return {op: bearer in ROLES, op in OPS |
[role,bearer] in INH*, [[op,object],bearer] in PR}

UserOperationsOnObject(user, object):
pre-condition: user in USERS, object in OBJS;
return {op: heir in ROLES, bearer in ROLES, op in OPS |
[user,heir] in UR, [heir,bearer] in INH*,
[[op,object],bearer] in PR}

Sept. 2007 Scott Stoller, Stony Brook University 29

Static Separation of Duty (SSD)

SsdNAMES: set(SsdName)
SsdNR: set(tuple(SsdName, Role))
SsdNC: set(tuple(SsdName, int)) //cardinality
The cardinality must be in the range 1 .. #rs-1.

SSD Requirement for core RBAC with SSD constraints:
forall u in USERS, [name,n] in SsdNC |
#{r: r in AssignedRoles(u) | [name,r] in SsdNR} <= n

SSD Requirement for hierarchical RBAC:
same except replace AssignedRoles with AuthorizedRoles

Sept. 2007 Scott Stoller, Stony Brook University 30

Functional Spec. for core RBAC with SSD

Core RBAC with SSD constraints extends core RBAC. AssignedUser is redefined to also check that the updated user assignment would satisfy the SSD constraints.

New Administrative Commands:
Create/DeleteSsdSet, Add/DeleteSsdRoleMember, SetSsdSetCardinality

Non-deletion administrative commands check that the new or updated SSD constraint would be satisfied, and that the cardinality would be in the required range.

New Review Functions
SsdRoleSets, SsdRoleSetRoles, SsdRoleSetCardinality

Sept. 2007 Scott Stoller, Stony Brook University 31

Functional Spec. for Hierarchical RBAC with SSD

General hierarchical RBAC with SSD is defined similarly to core RBAC with SSD, except:

1. It extends general hierarchical RBAC and redefines also AddInheritance to check that the SSD constraints would be satisfied.
2. It also extends core RBAC with SSD constraints and redefines the non-deletion new administrative commands to use AuthorizedRoles in place of AssignedRoles.

Sept. 2007 Scott Stoller, Stony Brook University 32

Dynamic Separation of Duty (DSD)

DsdNAMES: set(DsdName)
DsdNR: set(tuple(DsdName, Role))
DsdNC: set(tuple(DsdName, int))

The cardinality must be in the range $1 \dots \#rs-1$.

DSD Requirement (same for core and hierarchical RBAC):

forall s in SESSIONS, $[name,n]$ in DsdNC |
 $\#\{r: r \text{ in SessionRoles}(s) \mid [name,r] \text{ in DsdNR}\} \leq n$

Sept. 2007 Scott Stoller, Stony Brook University 33

Functional Spec. for core RBAC with DSD

Core RBAC with DSD extends core RBAC.

New administrative commands are added to create, modify, and delete DSD constraints, just as for SSD above, except that DSD constraints are checked instead of SSD constraints.

Supporting system functions CreateSession and AddActiveRole are redefined to also check that the DSD constraints would be satisfied.

New review functions are introduced to query DSD constraints.

Sept. 2007 Scott Stoller, Stony Brook University 34

Functional Spec. for Hierarchical RBAC with DSD

General hierarchical RBAC with DSD constraints is defined similarly to core RBAC with DSD, except:

1. It extends general hierarchical RBAC, inheriting all definitions unchanged.
2. It extends core RBAC with DSD constraints and redefines CreateSession and AddActiveRole to use AuthorizedRoles in place of AssignedRoles (when checking whether the user is authorized for roles; this does not affect DSD directly).

Sept. 2007 Scott Stoller, Stony Brook University 35