Information Security

Van K Nguyen - HUT

Access Control

Topics

- Overview
- Access Control Matrix model
- Discretionary Access Control (DAC)
- Mandatory Access Control (MAC) and an example model
- Role Based Access Control (RBAC)
- Access Control in Unix

What is AC

- Quote from Ross Anderson (text “Security Engineering”)
  - Its function is to control which principals (persons, processes, machines, …) have access to which resources in the system -- which files they can read, which programs they can execute, and how they share data with other principals, and so on.

Access Control is Pervasive

- Application
  - business applications
- Middleware
  - DBMS
- Operating System
  - controlling access to files, ports
- Hardware
  - memory protection, privilege levels

Access Control Matrix – A general model for protection systems

- Lampson'1971
  - "Protection"
- Refined by Graham and Denning'1972
  - "Protection---Principles and Practice"
- Harrison, Ruzzo, and Ullman'1976
  - "Protection in Operating Systems"

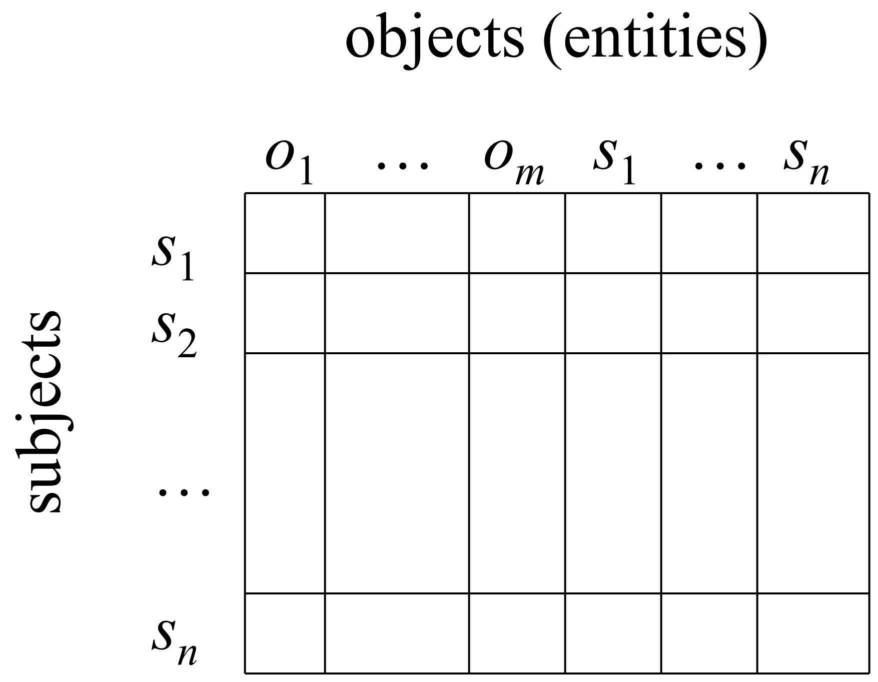
Overview

- Protection state of system
  - Describes current settings, values of system relevant to protection
- Access control matrix
  - Describes protection state precisely
  - Matrix describing rights of subjects
  - State transitions change elements of matrix

Access Matrix

- A set of subjects S
- A set of objects O
- A set of rights R
- An access control matrix
  - one row for each subject
  - one column for each subject/object
  - elements are right of subject on another subject or object

Description



- Subjects $S = \{ s_1,\ldots,s_n \}$
- Objects $O = \{ o_1,\ldots,o_m \}$
- Rights $R = \{ r_1,\ldots,r_k \}$
- Entries $A[s_i, o_j] \subseteq R$
- $A[s_i, o_j] = \{ r_x, \ldots, r_y \}$ means subject s_i has rights $r_x, \ldots, r_y$ over object o_j

Example 1

- Processes *p*, *q*
- Files *f*, *g*
- Rights *r*, *w*, *x*, *a*, *o*

	f	*g*	*p*	*q*
p	*rwo*	*r*	*rwxo*	*w*
q	*a*	*ro*	*r*	*rwxo*

Example 2

- Procedures *inc_ctr*, *dec_ctr*, *manage*
- Variable *counter*
- Rights +, –, *call*

	counter	*inc_ctr*	*dec_ctr*	*manage*
inc_ctr	+			
dec_ctr	–			
manage		*call*	*call*	*call*

Implementation

- Storing the access matrix
  - by rows: capability lists
  - by column: access control lists
  - through indirection:
    - e.g., key and lock list
    - e.g., groups, roles, multiple level of indirections, multiple locks
- How to do indirection correctly and conveniently is the key to management of access control.

Implementation

Access Control List (column) (ACL)

File 1	**File 2**
Joe:Read	Joe:Read
Joe:Write	Sam:Read
Joe:Own	Sam:Write
	Sam:Own

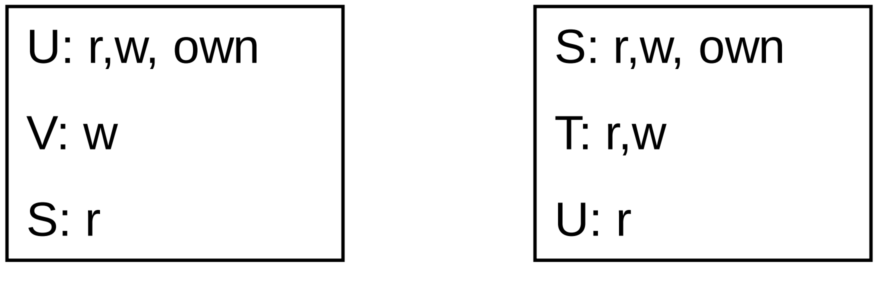
Capability List (row)

Joe: File 1/Read, File 1/Write, File 1/Own, File 2/Read
Sam: File 2/Read, File 2/Write, File 2/Own

Access Control Triples

Subject	Access	Object
Joe	Read	File 1
Joe	Write	File 1
Joe	Own	File 1
Joe	Read	File 2
Sam	Read	File 2
Sam	Write	File 2
Sam	Own	File 2

Access control lists



Object F Object G

- ACL is a list of permissions attached to an object
  - Who can modify the object's ACL?
  - What changes are allowed?
  - How are contradictory permissions handled?
  - How is revocation handled?

Owners and Groups

- Who can modify the object's ACL?
  - One way is by introducing owners of objects
- With ACLs we can define any combination of access, but that makes them difficult to manage
  - Group allow relatively fine-grained access control while making ACLs easier to manage
- Owners and groups can change

Capability lists

- One way to partition the matrix is by rows.
  - All access rights of one user together, stored in a data structure called a **capability list**
    - Lists all the access rights or capabilities that a user has.
    - E.g. Fred --> /dev/console(RW)--> fred/prog.c(RW)--> fred/letter(RW) --> /usr/ucb/vi(X) Jane --> /dev/console(RW)--> fred/prog.c(R)--> fred/letter() --> /usr/ucb/vi(X)

Capability lists

- All access to objects is done through capabilities
  - Every program holds a set of capabilities
  - Each program holds a small number of capabilities
  - The only way a program can obtain capabilities is to have them granted as a result of some communication
  - The set of capabilities held by each program must be as small as possible (*principle of least privilege)*
- Example: EROS Operating System
  - http://www.eros-os.org/eros.html

Harrison-Ruzzo-Ullman model

- **Discretionary Access Control**
  - Rights defined on specific (subject, object), decided by individual owners (as oppose to **Mandatory Access Control**, decided by system policies)
- HRU work
  - Formulating access matrices, towards Operating Systems
  - Provide a model that is sufficiently powerful to encode several access control approaches, and precise enough so that security properties can be analyzed
  - Introduce the "safety problem"
  - Show that the safety problem
    - is decidable in certain cases
    - is undecidable in general
    - is undecidable in monotonic case

Primitive Operations

- **create subject** *s*; **create object** *o*
  - Creates new row, column in ACM; creates new column in ACM
- **destroy subject** *s*; **destroy object** *o*
  - Deletes row, column from ACM; deletes column from ACM
- **enter** *r* **into** *A*[*s*, *o*]
  - Adds *r* rights for subject *s* over object *o*
- **delete** *r* **from** *A*[*s*, *o*]
  - Removes *r* rights from subject *s* over object *o*

Creating File

- Process *p* creates file *f* with *r* and *w* permission

```
command create•file(p, f)
    create object f;
    enter own into A[p, f];
    enter r into A[p, f];
    enter w into A[p, f];
end
```

Mono-Operational Commands

- Make process *p* the owner of file *g*

```
command make•owner(p, g)
    enter own into A[p, g];
end
```

- Mono-operational command
  - Single primitive operation in this command

Conditional Commands

- Let *p* give *q r* rights over *f*, if *p* owns *f*

```
command grant•read•file•1(p, f, q)
    if own in A[p, f]
    then
          enter r into A[q, f];
end
```

- Mono-conditional command
  - Single condition in this command

Discretionary Access Control (DAC)

- No precise definition
- Widely used in modern operating systems
- Often has the notion of owner of an object
- The owner controls other users' accesses to the object
- Allows access rights to be propagated to other subjects

Drawbacks in DAC

- DAC cannot protect against
  - Trojan horse
  - Malware
  - Software bugs
  - Malicious local users
- Cannot control information flow

Mandatory Access Control (MAC)

Mandatory Access Control

- *Objects:* security classification

  e.g., grades=(confidential, {student-info})
- *Subjects:* security clearances

  e.g., Joe=(confidential, {student-info})
- *Access rules:* defined by comparing the security classification of the requested objects with the security clearance of the subject

  e.g., subject can read object only if label(subject) dominates label(object)

Mandatory Access Control

- If *access control rules* are satisfied, access is permitted

  e.g., Joe wants to read grades.

  label(Joe)=(confidential,{student-info})

  label(grades)=(confidential,{student-info})

  Joe is permitted to read grades

- *Granularity* of access rights!

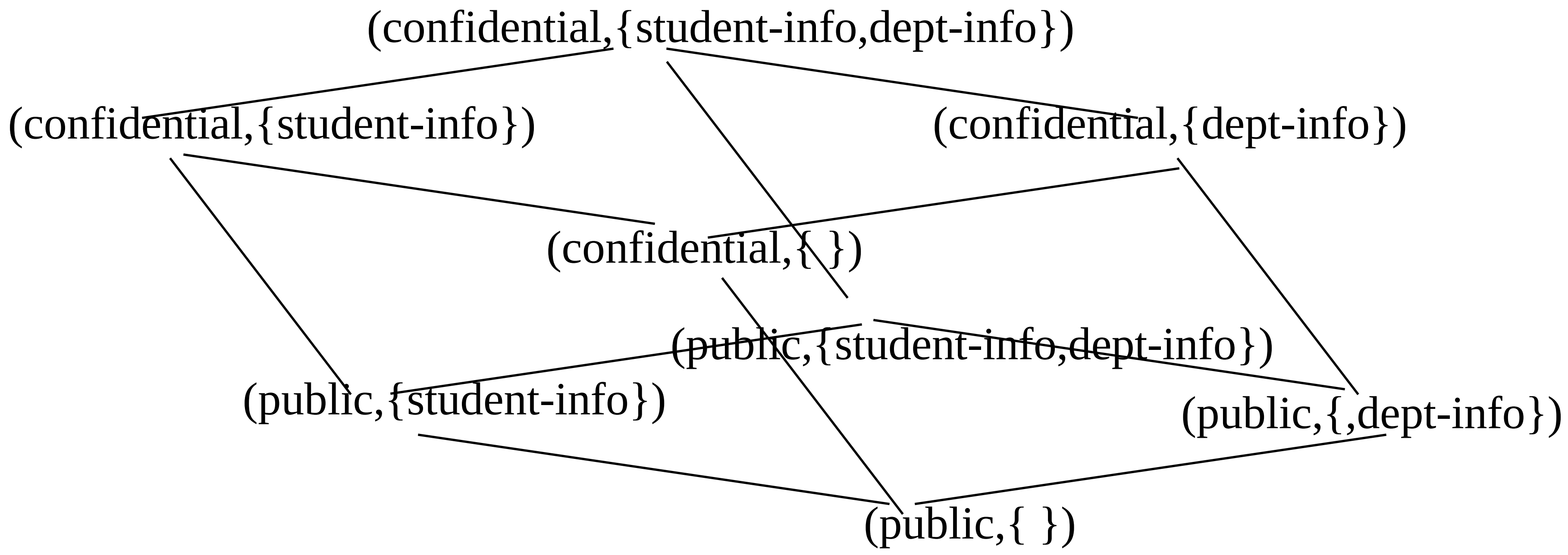
Mandatory Access Control

Security Classes (labels): (A,C)

A – total order authority level

C – set of categories

e.g., A = confidential > public , C = {student-info, dept-info}


(confidential,{student-info,dept-info})
(confidential,{student-info})
(confidential,{dept-info})
(confidential,{ })
(public,{student-info,dept-info})
(public,{student-info})
(public,{,dept-info})
(public,{ })

Mandatory Access Control

- *Dominance* (≥): label l=(A,C) dominates l'=(A',C') iff A ≥ A' and C ⊇ C'

  e.g., (confidential,{student-info}) ≥ (public,{student-info})

  BUT NOT

  (confidential, {student-info}) ≥ (public,{student-info, department-info})

Bell- LaPadula (BLP) Model

- Confidentiality protection
- Lattice-based access control
  - Subjects
  - Objects
  - Security labels
- Supports decentralized administration

BLP Reference Monitor

- All accesses are controlled by the reference monitor
- Cannot be bypassed
- Access is allowed iff the resulting system state satisfies all security properties
- *Trusted subjects*: subjects trusted not to compromise security

BLP Axioms 1.

Simple-security property: a subject *s* is allowed to read an object *o* *only if* the security label of *s* dominates the security label of *o*

- No read up
- Applies to *all subjects*

  Subject *s* can read object *o* iff $L(o) \leq L(s)$ and *s* has permission to read *o*

  - Note: combines mandatory control (relationship of security levels) and discretionary control (the required permission)

BLP Axioms 2.

**-property*: a subject *s* is allowed to write an object *o* *only if* the security label of *o* dominates the security label of *s*

- No write down
- Applies to *un-trusted subjects* only
  - Subject *s* can write object *o* iff $L(s) \leq L(o)$ and *s* has permission to write *o*
  - Note: combines mandatory control (relationship of security levels) and discretionary control (the required permission)

Example

security level	*subject*	*object*
Top Secret	Tamara	Personnel Files
Secret	Samuel	E-Mail Files
Confidential	Claire	Activity Logs
Unclassified	Ulaley	Telephone Lists

- Tamara can read all files
- Claire cannot read Personnel or E-Mail Files
- Ulaley can only read Telephone Lists

Levels and Lattices

- Security level is (*clearance*, *category set*)
  - (Top Secret, { NUC, EUR, ASI })
  - (Confidential, { EUR, ASI })
  - (Secret, { NUC, ASI })
- (A, C) *dom* (A', C') iff $A' \le A$ and $C' \subseteq C$
  - (Top Secret, {NUC, ASI}) *dom* (Secret, {NUC})
  - (Secret, {NUC, EUR}) *dom* (Confidential,{NUC, EUR})
  - (Top Secret, {NUC}) $\neg$*dom* (Confidential, {EUR})

MAC Overview

- Advantages:
  - Very secure
  - Centralized enforcement
- Disadvantages:
  - May be too restrictive
  - Need additional mechanisms to implement multi-level security system
  - Security administration is difficult

Role-Based Access Control (RBAC)

RBAC Motivation

- Multi-user systems
- Multi-application systems
- Permissions are associated with roles
- Role-permission assignments are persistent v.s. user-permission assignments
- Intuitive: competency, authority and responsibility

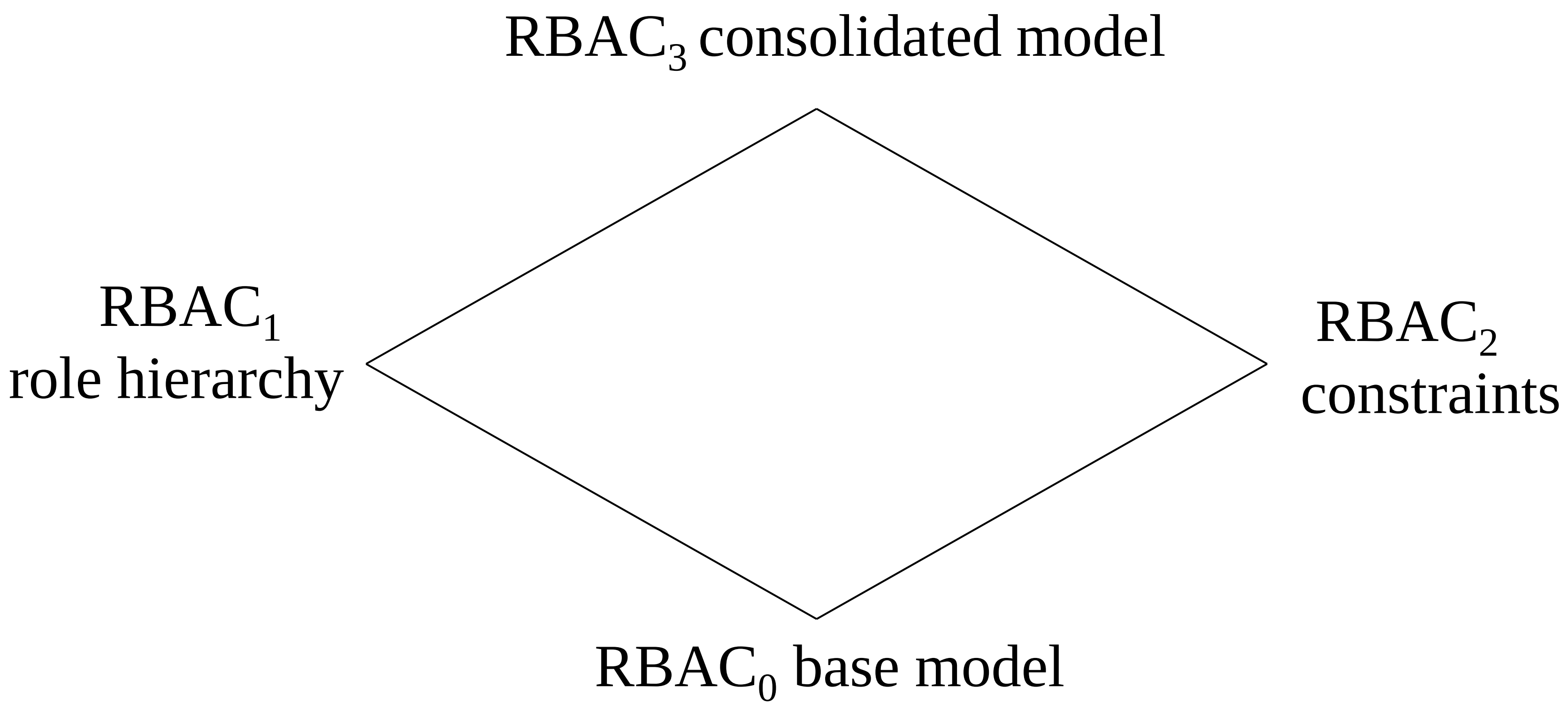
Motivation

- Express organizational policies
  - Separation of duties
  - Delegation of authority
- Flexible: easy to modify to meet new security requirements
- Supports
  - Least-privilege
  - Separation of duties
  - Data abstraction

Roles

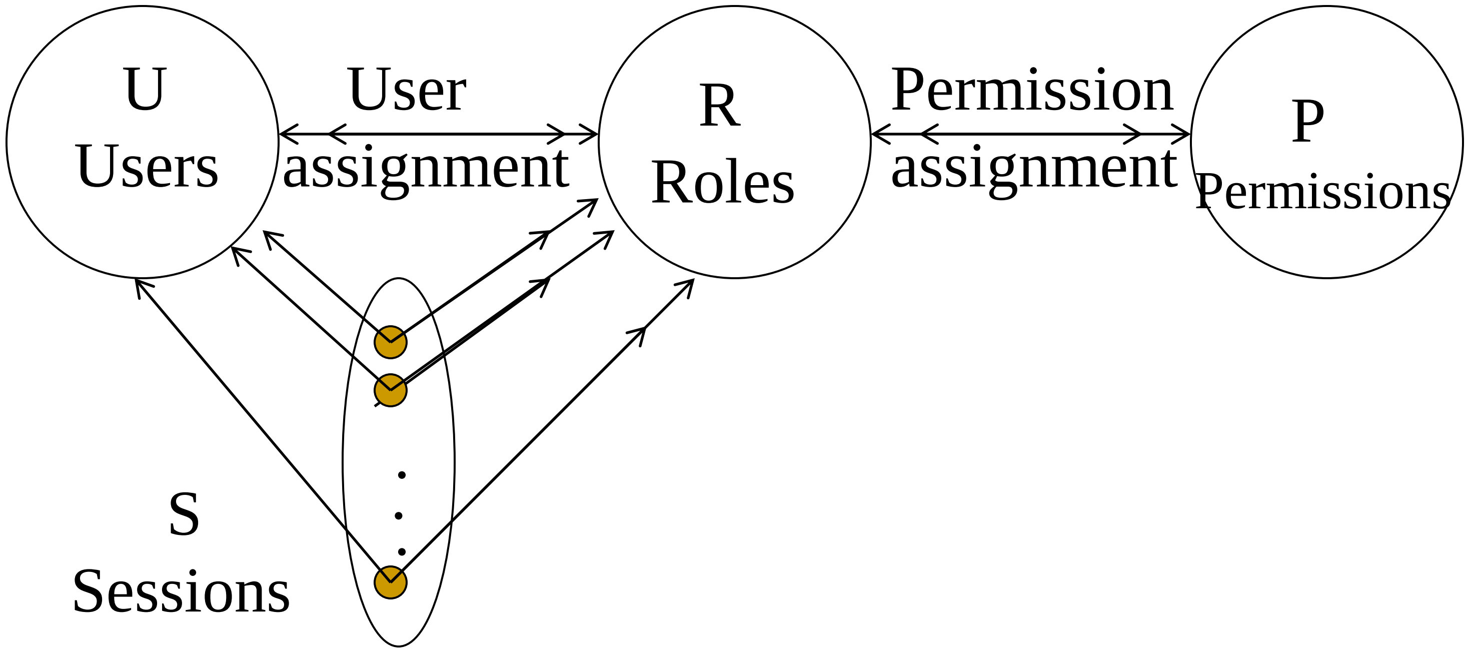
- **User group**: collection of user with possibly different permissions
- **Role**: mediator between collection of users and collection of permissions
- **RBAC** independent from DAC and MAC (they may coexist)
- RBAC is **policy neutral**: configuration of RBAC determines the policy to be enforced

RBAC



RBAC$_3$ consolidated model
RBAC$_1$ role hierarchy
RBAC$_2$ constraints
RBAC$_0$ base model

$RBAC_0$


U
Users
User
assignment
R
Roles
Permission
assignment
P
Permissions
S
Sessions

RBAC$_0$

- User: human beings
- Role: job function (title)
- Permission: approval of a mode of access
  - Always positive
  - Abstract representation
  - Can apply to single object or to many

RBAC$_0$

- UA: user assignments
  - Many-to-many
- PA: Permission assignment
  - Many-to-many
- Session: mapping of a user to possibly many roles
  - Multiple roles can be activated simultaneously
  - Permissions: union of permissions from all roles
  - Each session is associated with a single user
  - User may have multiple sessions at the same time

RBAC$_0$ Components

- **U**sers, **R**oles, **P**ermissions, **S**essions
- PA $\subseteq$ P x R (many-to-many)
- UA $\subseteq$ U x R (many-to-many)
- user: S → U, mapping each session s_i to a single user user(s_i)
- roles: S → 2^R, mapping each session s_i to a set of roles:
  - roles(s_i) $\subseteq$ {r | (user(s_i),r) $\in$ UA} and s_i has permissions $\cup_{r \in roles(s_i)}$ {p | (p,r) $\in$ PA}

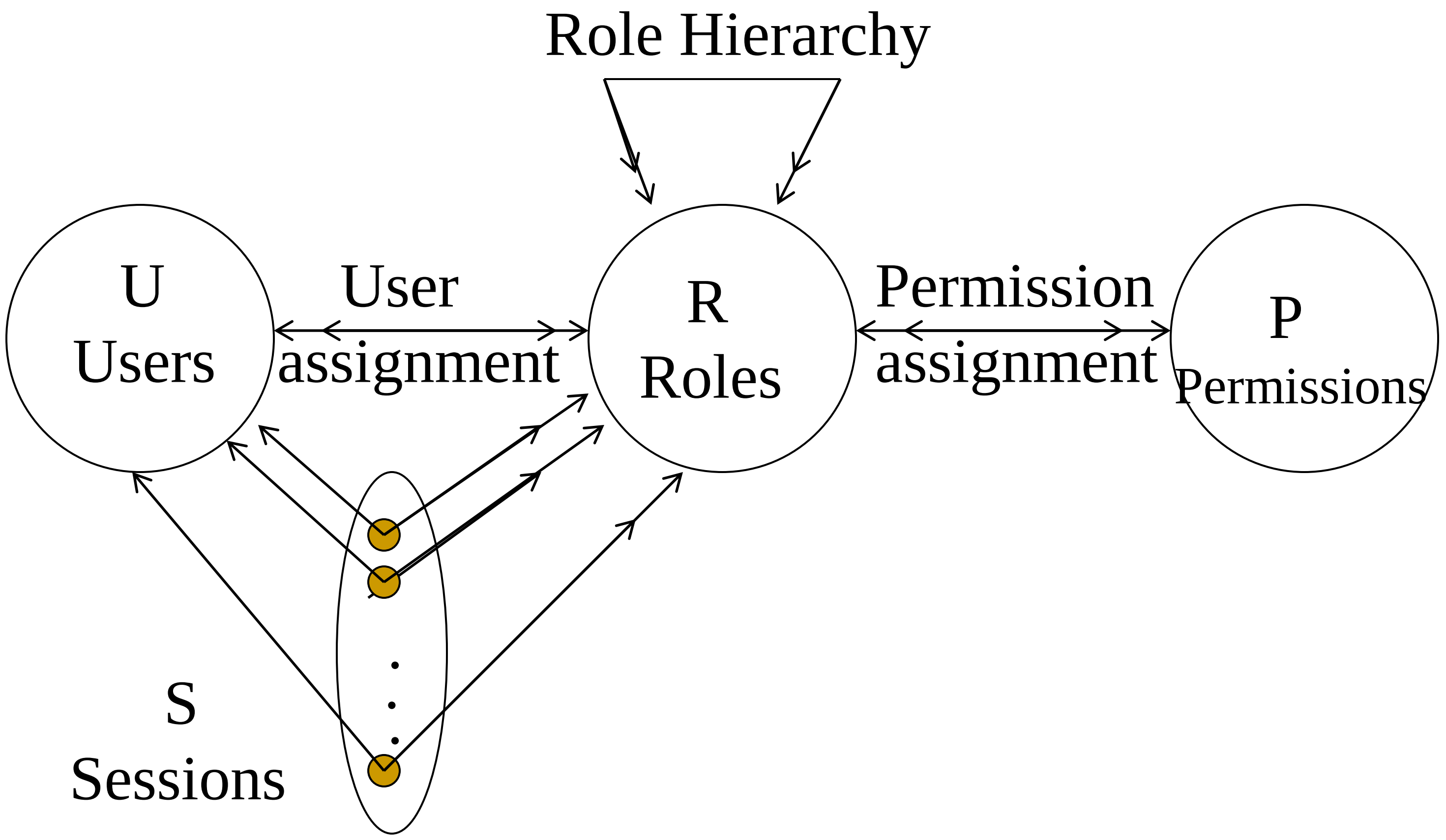
RBAC$_0$

- Permissions apply to data and resource objects only
- Permissions do NOT apply to RBAC components
- Administrative permissions: modify U,R,S,P
- Session: under the control of user to
  - Activate any subset of permitted roles
  - Change roles within a session

RBAC$_1$

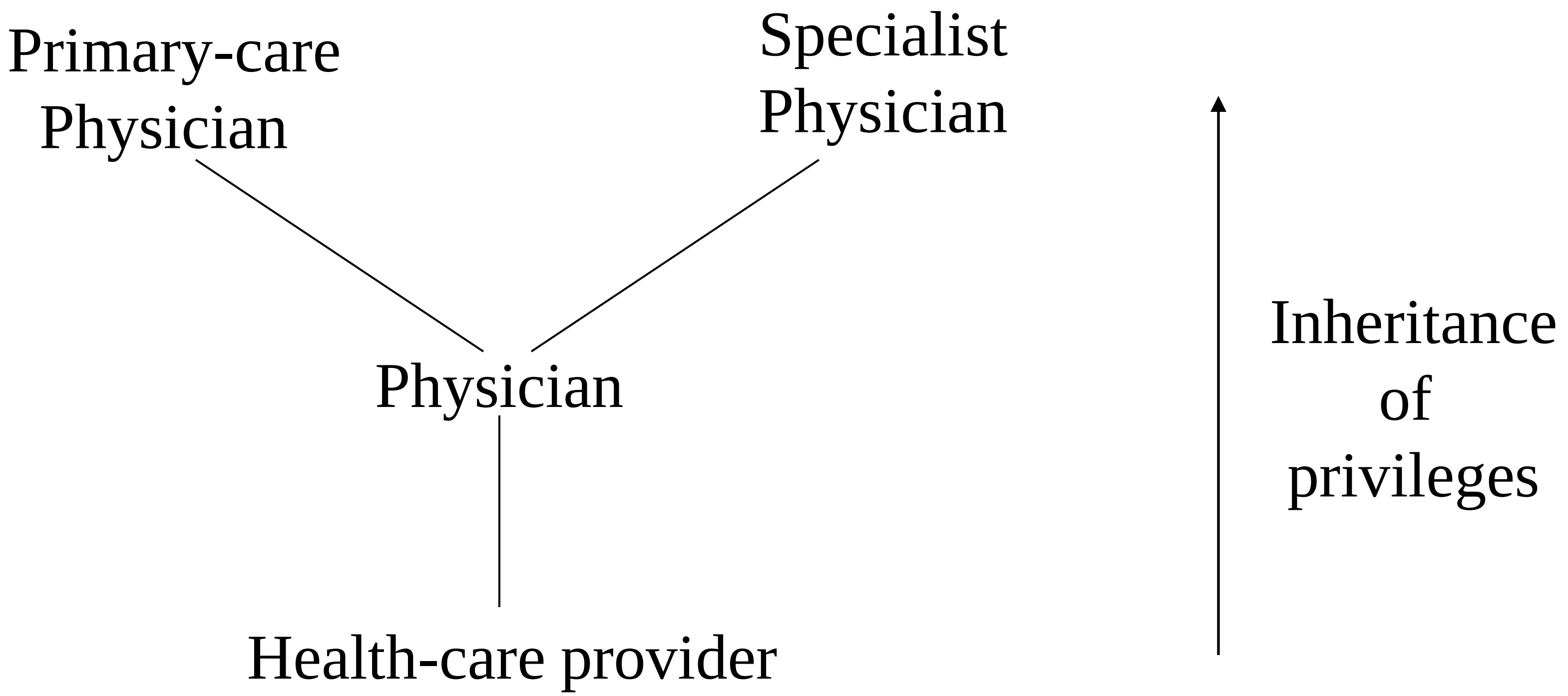
- Structuring roles
- Inheritance of permission from junior role (bottom) to senior role (top)
- Partial order
  - Reflexive
  - Transitive
  - Anti-symmetric

RBAC$_1$


Role Hierarchy
U
Users
User
assignment
R
Roles
Permission
assignment
P
Permissions
S
Sessions

RBAC$_1$

Role Hierarchy



RBAC$_1$ Components

- Same as RBAC$_0$: **U**sers, **R**oles, **P**ermissions, **S**essions, PA $\subseteq$ P x R, UA $\subseteq$ U x R, user: S $\rightarrow$ U, mapping each session s_i to a single user user(s_i)
- RH $\subseteq$ R x R, partial order ($\geq$ dominance)
- roles: S $\rightarrow$ 2^R, mapping each session s_i to a set of roles
  - roles(s_i) $\subseteq$ $\{r \mid (\exists r' \geq r)\ [(user(s_i), r') \in UA]\}$ and s_i has permissions $\cup_{r \in roles(s_i)} \{p \mid (\exists r'' \leq r)\ [(p, r'') \in PA]\}$

Access Control in Unix

General Concepts

- Users, Groups, Processes, Files
  - Each user has a unique UID
  - Each group has a unique GID
  - Each process has a unique PID
  - Users belong to multiple groups GID
  - Objects whose access is controlled
    - Files
    - Directories
- Organization of Objects
  - Files are arranged in a hierarchy
  - Files exist in directories
  - Directories are one type of files
  - In UNIX, access on directories are not inherited

Basic Permissions Bits on Files

- Permission:
  - Read: control reading the content of a file
  - Write: controls changing the content of a file
  - Execute: controls loading the file then execute
- Many operations can be performed only by the owner of the file
- Where are Permission Bits Kept?
  - Each file/directory has associated an i-node.
  - The file type, permissions, owner UID and owner GID are save on disk in the inode of a file or directory

Permission Bits on Directories

- Read: for showing file names in a directory
- Execution: for traversing a directory
  - does a lookup, allows one to find inode # from file name
  - ‘chdir’ to a directory requires execution
- Write + execution: for creating/deleting files in the directory
  - requires no permission on the file
- Accessing a file a path name: need execution permission to all directories along the path

The Three Sets of Permission Bits

- Permission example

  drwxr-xr-x

  - First: directory or not
  - Next three: owner permission
    - if the user is the owner of a file then the r/w/x bits for owner apply
  - Next three: group permission
    - if the user belongs to the group the file belongs to then the r/w/x bits for group apply
  - Next three: others permission
    - Apply when not the owner or belong to the group
- Where are Permission Bits Kept?
  - Each file/directory has associated an inode.
  - The file type, permissions, owner UID and owner GID are save on disk in the inode of a file or directory

Users vs. Subjects

- Permission bits talk about what users can access a file
  - ➔ but it is subjects (processes) to perform actions on files
  - When a subject accesses a file, the system check which user it is acting on behalf of
- Problem: what if an executable need stronger permission than the subject calling it
  - The **passwd** program needs to update a system-wide password file, which ordinary users should not be able to modify, but only root can modify
  - But remember, it needs to be run by ordinary users

Real User ID vs. Effective User ID

- Each process has three user IDs
  - real user ID (ruid): owner of the process
  - effective user ID (euid): used in most access control decisions, often the same as ruid unless there is a change
  - saved user ID (suid): keeps the previous euid if it was a change
- and three group IDs
  - real group ID
  - effective group ID
  - saved group ID

The setuid flag

- When used for a file
  - allows certain processes to have more than ordinary privileges while still being executable by ordinary users
  - When set, the effective uid of the calling process takes the value of the owner of the file

How the process user IDs work

- When a process is created by *fork*
  - it inherits all three UIDs from its parent process
- When a process executes a file by *exec*

  if (the setuid bit of the file is off)

      it keeps its three user IDs

  otherwise // the setuid is set

      euid of the process = ruid of the file

      suid = previous euid
- How to solve the passwd problem and the likes?
  - Passwd is owned by root and setuid is set
  - When a process executes it, then effective user becomes root, so the program runs as root on behalf of the user (only within the passwd work)
- Can be a security flaw if the mechanism for temporary higher privilege is abused