Introduction to
Computer Security

Access Control Matrix
Take-grant model

September 9, 2004

Protection System

- State of a system
  - Current values of
    - memory locations, registers, secondary storage, etc.
    - other system components
- Protection state (P)
  - A system state that is considered secure
- A protection system
  - Describes the conditions under which a system is secure (in a protection state)
  - Consists of two parts:
    - A set of generic rights
    - A set of commands
- State transition
  - Occurs when an operation (command) is carried out

Protection System

- Subject (S: set of all subjects)
  - Active entities that carry out an action/operation on other entities; Eg.: users, processes, agents, etc.
- Object (O: set of all objects)
  - Eg.:Processes, files, devices
- Right
  - An action/operation that a subject is allowed/disallowed on objects

Access Control Matrix Model

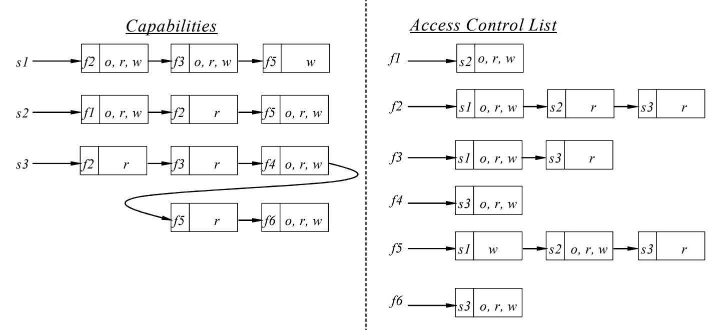
- Access control matrix
  - Describes the protection state of a system.
  - Characterizes the rights of each subject
  - Elements indicate the access rights that subjects have on objects
- ACM is an abstract model
  - Rights may vary depending on the object involved
- ACM is implemented primarily in two ways
  - Capabilities (rows)
  - Access control lists (columns)

Access Control Matrix

o: own
r: read
w:write

	f1	f2	f3	f4	f5	f6
s1		o, r, w	o, r, w		w	
s2	o, r, w	r			o, r, w	
s3		r	r	o, r, w	r	o, r, w

Access Matrix



Access Control Matrix

Hostnames	*Telegraph*	*Nob*	*Toadflax*
Telegraph	*own*	*ftp*	*ftp*
Nob		*ftp, nsf, mail, own*	*ftp, nfs, mail*
Toadflax		*ftp, mail*	*ftp, nsf, mail, own*

•*telegraph* is a PC with ftp client but no server

•*nob* is provides NFS but not to Toadfax

•*nob* and *toadfax* can exchange mail

	Counter	*Inc_ctr*	*Dcr_ctr*	*Manager*
Inc_ctr	+			
Dcr_ctr	-			
manager		*Call*	*Call*	*Call*

Boolean Expression Evaluation

- ACM controls access to database fields
  - Subjects have attributes
  - Verbs define type of access
  - Rules associated with objects, verb pair
- Subject attempts to access object
  - Rule for object, verb evaluated, grants or denies access

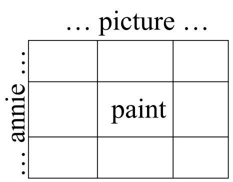
Example

- Subject annie
  - Attributes role (artist), groups (creative)
- Verb paint
  - Default 0 (deny unless explicitly granted)
- Object picture
  - Rule:

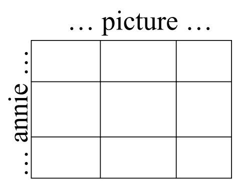
    paint: 'artist' in subject.role and
    'creative' in subject.groups and
    time.hour $\geq$ 0 and time.hour < 5

ACM at 3AM and 10AM

At 3AM, time condition met; ACM is:



At 10AM, time condition not met; ACM is:



Access Controlled by History

- Statistical databases need to
  - answer queries on groups
  - prevent revelation of individual records
- Query-set-overlap control
  - Prevent an attacker to obtain individual piece of information using a set of queries C
  - A parameter *r* (=2) is used to determine if a query should be answered

Name	Position	Age	Salary
Alice	Teacher	45	40K
Bob	Aide	20	20K
Cathy	Principal	37	60K
Dilbert	Teacher	50	50K
Eve	Teacher	33	50K

Access Controlled by History

- Query 1:
  - sum_salary(position = teacher)
  - Answer: 140K
- Query 2:
  - sum_salary(age > 40 & position = teacher)
  - Should not be answered as Matt's salary can be deduced
- Can be represented as an ACM

Name	Position	Age	Salary
Celia	Teacher	45	40K
Leonard	Teacher	50	50K
Matt	Teacher	33	50K

Name	Position	Age	Salary
Celia	Teacher	45	40K
Leonard	Teacher	50	50K

Solution: Query Set Overlap Control (Dobkin, Jones & Lipton '79)

- Query valid if intersection of query coverage and each previous query < *r*
- Can represent as access control matrix
  - Subjects: entities issuing queries
  - Objects: *Powerset* of records
  - $O_s(i)$: objects referenced by *s* in queries *1..i*
  - *A*[*s*,*o*] = read iff $\forall_{q \in O_s(i-1)} |q \cap o| < r$

- Query 1: O_1 = {Celia, Leonard, Matt} so the query can be answered. Hence
  - A[asker, Celia] = {read}
  - A[asker, Leonard] = {read}
  - A[asker, Matt] = {read}
- Query 2: O_2 = {Celia, Leonard} but $| O_2 \cap O_1 | = 2$; so the query cannot be answered
  - A[asker, Celia] = ∅
  - A[asker, Leonard] = ∅

State Transitions

- Let initial state $X_0 = (S_0, O_0, A_0)$
- Notation
  - $X_i \vdash_{\tau_{i+1}} X_{i+1}$: upon transition τ_{i+1}, the system moves from state X_i to X_{i+1}
  - $X \vdash^* Y$: the system moves from state X to Y after a set of transitions
  - $X_i \vdash c_{i+1}(p_{i+1,1}, p_{i+1,2}, \ldots, p_{i+1,m}) X_{i+1}$: state transition upon a command
- For every command there is a sequence of state transition operations

Primitive commands (HRU)

Create subject s	Creates new row, column in ACM;
Create object o	Creates new column in ACM
Enter r into $a[s, o]$	Adds *r* right for subject *s* over object *o*
Delete r from $a[s, o]$	Removes *r* right from subject *s* over object *o*
Destroy subject s	Deletes row, column from ACM;
Destroy object o	Deletes column from ACM

Create Subject

- Precondition: $s \notin S$
- Primitive command: **create subject** *s*
- Postconditions:
  - $S' = S \cup \{ s \}$, $O' = O \cup \{ s \}$
  - $(\forall y \in O')[a'[s, y] = \emptyset]$ (row entries for s)
  - $(\forall x \in S')[a'[x, s] = \emptyset]$ (column entries for s)
  - $(\forall x \in S)(\forall y \in O)[a'[x, y] = a[x, y]]$

Create Object

- Precondition: $o \notin O$
- Primitive command: **create object** *o*
- Postconditions:
  - $S' = S$, $O' = O \cup \{ o \}$
  - $(\forall x \in S')[a'[x, o] = \varnothing]$ (column entries for *o*)
  - $(\forall x \in S)(\forall y \in O)[a'[x, y] = a[x, y]]$

Add Right

- Precondition: $s \in S$, $o \in O$
- Primitive command: enter *r* into $a[s, o]$
- Postconditions:
  - $S' = S$, $O' = O$
  - $a'[s, o] = a[s, o] \cup \{ r \}$
  - $(\forall x \in S' - \{ s \})(\forall y \in O' - \{ o \})$ $[a'[x, y] = a[x, y]]$

Delete Right

- Precondition: $s \in S$, $o \in O$
- Primitive command: **delete** r **from** $a[s, o]$
- Postconditions:
  - $S' = S$, $O' = O$
  - $a'[s, o] = a[s, o] - \{ r \}$
  - $(\forall x \in S' - \{ s \})(\forall y \in O' - \{ o \})$ $[a'[x, y] = a[x, y]]$

Destroy Subject

- Precondition: $s \in S$
- Primitive command: **destroy subject** s
- Postconditions:
  - $S' = S - \{ s \}$, $O' = O - \{ s \}$
  - $(\forall y \in O')[a'[s, y] = \varnothing]$ (row entries removed)
  - $(\forall x \in S')[a'[x, s] = \varnothing]$ (column entries removed)
  - $(\forall x \in S')(\forall y \in O')$ $[a'[x, y] = a[x, y]]$

Destroy Object

- Precondition: $o \in o$
- Primitive command: **destroy object** *o*
- Postconditions:
  - $S' = S$, $O' = O - \{ o \}$
  - $(\forall x \in S')[a'[x, o] = \varnothing]$ (column entries removed)
  - $(\forall x \in S')(\forall y \in O')\ [a'[x, y] = a[x, y]]$

System commands using primitive operations

- process *p* creates file *f* with owner *read* and *write* (*r*, *w*) will be represented by the following:

  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

- Defined commands can be used to update ACM

  Command *make_owner*(*p*, *f*)
  Enter *own* into *a*[*p*,*f*]
  End

- Mono-operational: the command invokes only one primitive

Conditional Commands

- Mono-operational + mono-conditional

```
Command grant_read_file(p, f, q)
  If own in a[p,f]
  Then
    Enter r into a[q,f]
End
```

- Mono-operational + biconditional

```
Command grant_read_file(p, f, q)
  If r in a[p,f] and c in a[p,f]
  Then
    Enter r into a[q,f]
End
```

- Why not "OR"??

Attenuation of privilege

- Principle of attenuation
  - A subject may not give rights that it does not posses to others
- Copy
  - Augments existing rights
  - Often attached to a right, so only applies to that right
    - *r* is read right that cannot be copied
    - *rc* is read right that can be copied Also called the *grant* right
- Own
  - Allows adding or deleting rights, and granting rights to others
  - Creator has the *own* right
  - Subjects may be granted *own* right
  - Owner may give rights that he does not have to others on the objects he owns (chown command)
    - Example: John owns file *f* but does not have *read* permission over it. John can grant *read* right on *f* to Matt.

Fundamental questions

- How can we determine that a system is secure?
  - Need to define what we mean by a system being "secure"
- Is there a generic algorithm that allows us to determine whether a computer system is secure?

What is a secure system?

- A simple definition
  - A secure system doesn't allow violations of a security policy
- Alternative view: based on distribution of rights to the subjects
  - Leakage of rights: (unsafe with res
    - Assume that *A* representing a secure state does not contain a right r in any element of A.
    - A right r is said to be leaked, if a sequence of operations/commands adds *r* to an element of *A,* which not containing *r*
- Safety of a system with initial protection state X_o
  - Safe with respect to r: System is *safe with respect to r* if *r* can never be leaked
  - Else it is called unsafe with respect to right r.

Safety Problem: *formally*

- Given
  - initial state $X_0 = (S_0, O_0, A_0)$
  - Set of primitive commands *c*
  - *r* is not in $A_0[s, o]$
- Can we reach a state X_n where
  - $\exists s,o$ such that $A_n[s,o]$ includes a right *r* not in $A_0[s,o]$?

  - If so, the system is not safe
  - But is "safe" secure?

Decidability Results *(Harrison, Ruzzo, Ullman)*

- Theorem: Given a system where each command consists of a single *primitive* command (mono-operational), there exists an algorithm that will determine if a protection system with initial state X_0 is safe with respect to right *r*.
- Proof: determine minimum commands *k* to leak
  - Delete/destroy: Can't leak (or be detected)
  - Create/enter: new subjects/objects "equal", so treat all new subjects as one
    - No test for absence
    - Tests on $A[s_1, o_1]$ and $A[s_2, o_2]$ have same result as the same tests on $A[s_1, o_1]$ and $A[s_1, o_2] = A[s_2, o_2] \cup A[s_2, o_2]$
  - If *n* rights leak possible, must be able to leak $n(|S_0|+1)(|O_0|+1)+1$ commands
  - Enumerate all possible states to decide

Decidability Results
(Harrison, Ruzzo, Ullman)

- It is undecidable if a given state of a given protection system is safe for a given generic right
- For proof – need to know Turing machines and halting problem

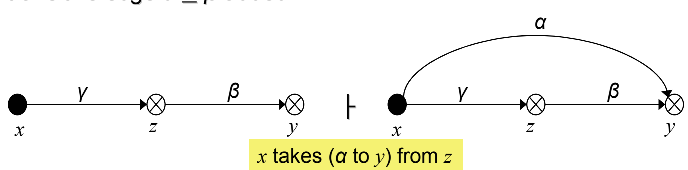
What is the implication?

- Safety decidable for some models
  - Are they practical?
- Safety only works if maximum rights known in advance
  - Policy must specify all rights someone could get, not just what they have
  - Where might this make sense?
- Next: Example of a decidable model
  - Take-Grant Protection Model

Take-Grant Protection Model

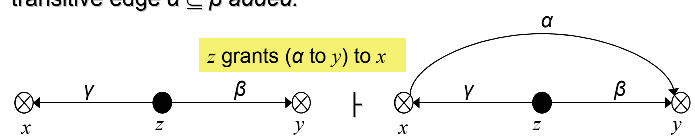
- System is represented as a directed graph
  - Subject: ●
  - Object: ○
  - Either: ⊗
  - Labeled edge indicate the rights that the source object has on the destination object
- Four graph rewriting rules ("de jure", "by law", "by rights")
  - The graph changes as the protection state changes according to

1. Take rule: if $t \in \gamma$, the take rule produces another graph with a transitive edge $\alpha \subseteq \beta$ added.

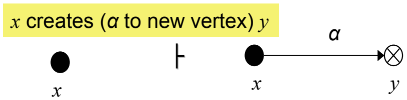


Take-Grant Protection Model

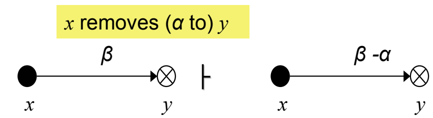
2. Grant rule: if $g \in \gamma$, the take rule produces another graph with a transitive edge $\alpha \subseteq \beta$ *added*.



3. Create rule:



4. Remove rule:

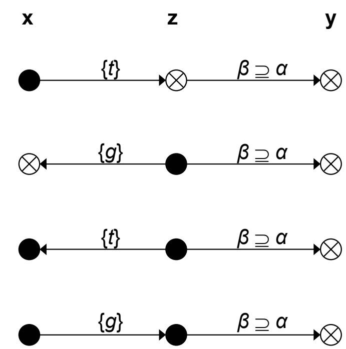


Take-Grant Protection Model: Sharing

- Given G_0, can vertex **x** obtain α rights over **y**?
  - Can_share(α,x, y,G_0) is true iff
    - $G_0 \vdash^* G_n$ using the four rules, &
    - There is an α edge from x to y in G_n
- *tg-path*: $\mathbf{v}_0,\ldots,\mathbf{v}_n$ with t or g edge between any pair of vertices $\mathbf{v}_i$, $\mathbf{v}_{i+1}$
  - Vertices *tg-connected* if *tg-path* between them
- Theorem: Any two subjects with *tg-path* of length 1 can share rights

Any two subjects with *tg-path* of length 1 can share rights

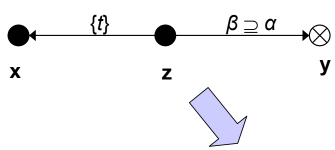
Can_share(α, x, y,G_0)

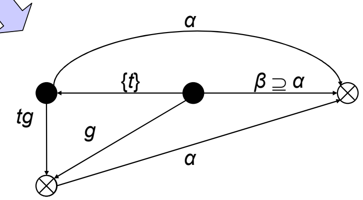


- Four possible length 1 *tg*-paths
  1. Take rule
  2. Grant rule
  3. Lemma 3.1
  4. Lemma 3.2

Any two subjects with *tg-path* of length 1 can share rights

Can_share(α, x, y, G_0)

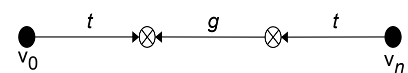




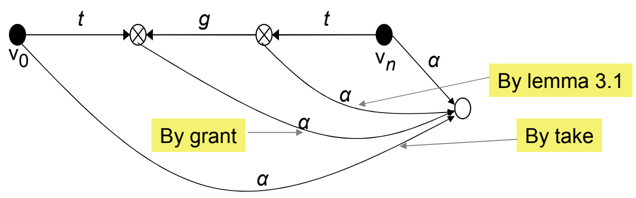
- Lemma 3.1
  - Sequence:
    - Create
    - Take
    - Grant
    - Take

Other definitions

- Island: Maximal *tg*-connected subject-only subgraph
  - Can_share all rights in island
  - Proof: Induction from previous theorem
- Bridge: *tg*-path between subjects v_0 and v_n with edges of the following form:
  - $t_{\rightarrow}^*$, $t_{\leftarrow}^*$
  - $t_{\rightarrow}^*$, $g_{\rightarrow}$, $t_{\leftarrow}^*$
  - $t_{\rightarrow}^*$, $g_{\leftarrow}$, $t_{\leftarrow}^*$



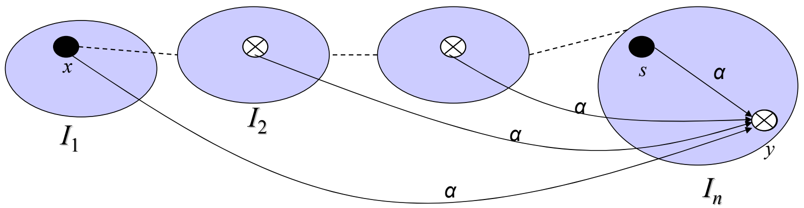
Bridge



Theorem: Can_share(α,**x**,**y**,G_0) (for subjects)

- Subject_can_share(α, x, y, G_0) is true iff if x and y are subjects and
  - there is an α edge from x to y in G_0

  OR if:
  - ∃ a subject s ∈ G_0 with an s-to-y α edge, and
  - ∃ islands I_1, …, I_n such that $x \in I_1$, **s** $\in I_n$, and there is a bridge from I_j to I_{j+1}



What about objects? Initial, terminal spans

- $\boldsymbol{x}$ *initially spans* to $\boldsymbol{y}$ if $\boldsymbol{x}$ is a subject and there is a *tg*-path between them with *t* edges ending in a *g* edge (i.e., $t_{\rightarrow}{}^{*}g_{\rightarrow}$)
  - $\boldsymbol{x}$ can grant a right to $\boldsymbol{y}$
- $\boldsymbol{x}$ *terminally spans* to $\boldsymbol{y}$ if $\boldsymbol{x}$ is a subject and there is a *tg*-path between them with *t* edges (i.e., $t_{\rightarrow}{}^{*}$)
  - $\boldsymbol{x}$ can take a right from $\boldsymbol{y}$

Theorem: Can_share(α,**x**,**y**,G_0)

- Can_share(α,x, y,G_0) iff there is an α edge from x to y in G_0 or if:
  - ∃ a vertex $s \in G_0$ with an s to y α edge,
  - ∃ a subject x' such that $x'=x$ or x' *initially spans* to x,
  - ∃ a subject s' such that $s'=s$ or s' *terminally spans* to s, and
  - ∃ islands I_1, …, I_n such that $x' \in I_1$, $s' \in I_n$, and there is a bridge from I_j to I_{j+1}