Controllo degli Accessi

Parte I

Indice

Capitolo 1

Introduzione

Il **controllo degli accessi** valuta l'accesso richiesto alle risorse dagli utenti autenticati e, sulla base di *regole di accesso* (definite all'interno) del sistema, determina se l'accesso sia garantito o negato.
Si occupa solamente dell'**accesso diretto**. Si basa su due concetti:

- **Autenticazione/Identificazione** dell'utente che fa la richiesta
  - importante anche per le problematiche di *accountability*; posso analizzare i log per capire chi ha fatto che cosa nel caso ci sia un problema
- **Correttezza delle autorizzazioni** con cui l'accesso viene valutato

1.1 Politiche, modelli, meccanismi

È utile fare una distinzione tra:

- **Politiche:** sono i requisiti di protezione ad alto livello che voglio applicare al mio sistema
- **Model:** viene usato per rappresentare la politica
- **Meccanismi:** implementano la politica con *hw* e *sw*

Risulta utile fare questa distinzione perché comporta dei vantaggi: posso verificare se il modello è corretto rispetto alla politica che ho definito; lo stesso meccanismo può essere usato per implementare politiche o modelli diversi.

1.1.1 Meccanismo

In letteratura prende il nome di *reference monitor*, deve soddisfarre le seguenti proprietà:

- non può essere modificabile; nel caso in cui venga fatto me ne devo accorgere

- non può essere bypassabile
- deve essere confinato ad una specifica parte del mio sistema (non distribuito)
- deve essere abbastanza piccolo per essere soggetto a processi di verifica formale

Il meccanismo deve essere sicuro rispetto ai canali di comunicazione non legittimi:

- **Storage channels:** le parti di memoria, prima di essere rese disponibili ad altri dati, dovrebbero essere *pulite* (se cancello un dato non è che *sparisce* dalla memoria fisica dal computer ...)
- **Covert channels:** canali non intesi per il trasferimento di informazioni che possono essere usati per inferire informazioni

Alcuni principi di design

- *Separazione dei privilegi:* non dare troppo *potere* ad un solo utente
- *Privilegio minimo:* voglio darti il minimo privilegio di cui hai bisogno

1.2 Processo di sviluppo di un AC

Una volta definito il modello, posso verificare due aspetti:

- **Completezza:** verificare che hai rappresentato tutti i requisiti di sicurezza della politica
- **Consistenza:** dev essere privo di contraddizioni (un utente ha sia accesso/negazione per una risorsa)

Capitolo 2

Discretionary (DAC) policies: approcci base

Sono politiche basate su:

- **identità** degli utenti
- definizione di regole di accesso (**autorizzazioni**), che stabiliscono *chi può fare che cosa*

Definite *discrezionale* perché gli utenti che sono proprietari dei dati possono amministrarli come vogliono, *a loro discrezione*; tipicamente, non ho un unico amministratore, ma ci sono più amministratori proprietari delle risorse: è in mano a qualcuno stabilire chi può accedere o meno alle risorse (non è escluso avere un unico amministratore).

2.1 Un esempio di modello

Si usa la *matrice degli accessi*, è una rappresentazione astratta della politica di protezione del sistema.
Formalmente, è caratterizzato da una tripla (S, O, A) che rappresenta lo stato del sistema, dove:

- S è il set degli utenti
- O è il set delle risorse, dove $S \subset O$ (un soggetto può essere anche un processo, e può essere anche una risorsa ...)
- A è la matrice, dove:
  - le righe corrispondono ai soggetti
  - le colonne corrispondono agli oggetti
  - $A[s, o]$ riporta i privilegi di s su o

	File 1	File 2	File 3	Program 1
Ann	own read write	read write		execute
Bob	read		read write	
Carl		read		execute read

I cambi di stato del sistema vengono fatti con dei comandi che chiamano delle **operazioni primitive**:

- `enter` r into $A[s, o]$
- `delete` r from $A[s, o]$
- `create` subject s
- ...

Sono della forma:

command $c(x_1, \ldots, x_k)$
  if r_1 in $A[x_{s_1}, x_{o_1}]$ and
    r_2 in $A[x_{s_2}, x_{o_2}]$ and
    .....
    $r_m$in $A[x_{s_m}, x_{o_m}]$
  then op_1
    op_2
    .....
    op_n
end

Un esempio:

command CREATE(subj,file)
  create object file
  enter Own *into* A[subj,file] **end.**

command CONFER$_{\text{read}}$(owner,friend,file)
  if Own in A[owner,file]
    then *enter* Read *into* A[friend,file] **end.**

command REVOKE$_{\text{read}}$(subj,exfriend,file)
  if Own in A[subj,file]
    then *delete* Read *from* A[exfriend,file] **end.**

2.2 Trasferimento dei privilegi

Il proprietario dei dati può dare il privilegio anche ad altri utenti. Può essere rappresentato in modo formale in due modi differenti:

- **Copy flag (*):** il soggetto trasferisce il privilegio ad altri; mantiene il privilegio

```
command TRANSFER_read(subj,friend,file)
        if Read* in A[subj,file]
          then enter Read into A[friend,file] end.
```

- **Transfer-only flag(+):** il soggetto trasferisce ad altri il privilegio ma perde l'autorizzazione

```
command TRANSFER-ONLY_read(subj,friend,file)
        if Read+ in A[subj,file]
          then delete Read+ from A[subj,file]
               enter Read+ into A[friend,file] end.
```

Partendo da uno stato *sicuro*, non deve accadere che applicando una o più operazioni si finisca in uno stato non sicuro.

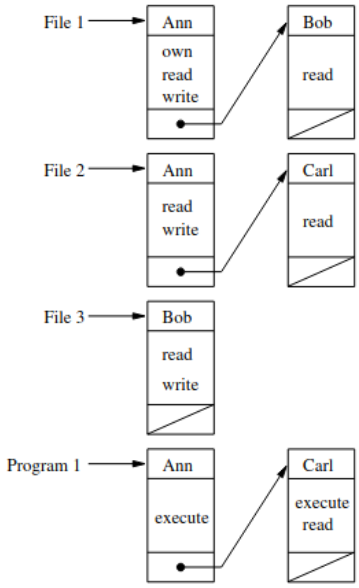
2.3 Implementazione della matrice

La matrice è spesso sparsa, salvarla sarebbe uno spreco di memoria. Ci sono diversi approcci alternativi:

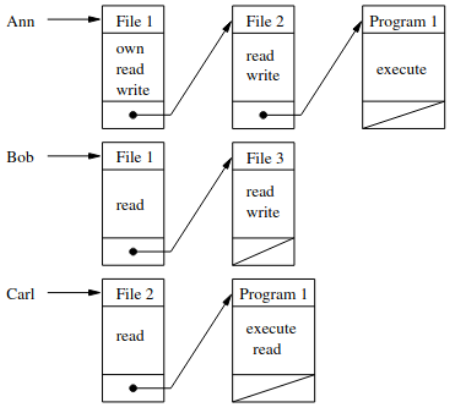
- **Tabella di autorizzazione**; tabella di tuple (S, O, A) non nulle

User	Access mode	Object
Ann	own	File 1
Ann	read	File 1
Ann	write	File 1
Ann	read	File 2
Ann	write	File 2
Ann	execute	Program 1
Bob	read	File 1
Bob	read	File 2
Bob	write	File 2
Carl	read	File 2
Carl	execute	Program 1
Carl	read	Program 1

- **Access Control Lists (ACLs)**: store by column; ad ogni risorsa associo una lista che mi dice gli utenti quali operazioni possono fare



- *Capability lists*; store by row; sono come le precedenti ma vengono fatti storando per utenti invece che per risorse. Sono state soppraffatte dalle ACLs



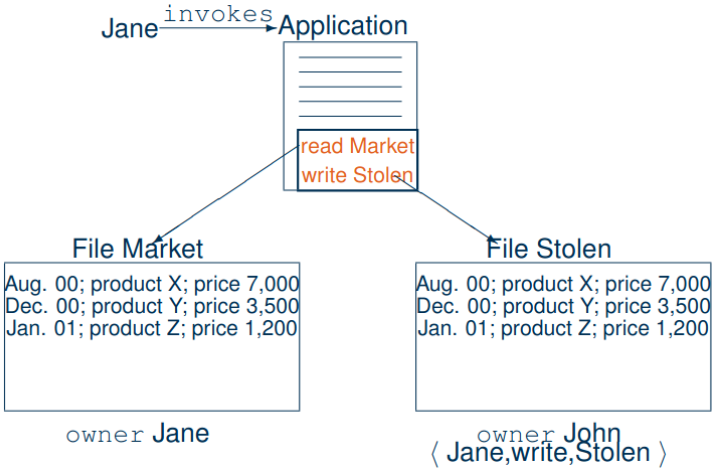
ACLs vs Capability lists

- non richiedono autenticazione del soggetto, ma richiedono la possibilità di verificare che non siano state impropriamente modificate . . . difficile da verificare (per questo non hanno avuto grande successo)

- le ACLs funzionano meglio quando fare delle operazioni di revoca per oggetto (ovviamente viceversa se devo fare revoche per soggetto)

2.4 Debolezze di DAC

Consentono il controllo solo sull'accesso **diretto**. Sono vulnerabili ai *trojan horses*, ovvero accessi indiretti.



L'idea è che vengono lasciate delle *operazioni nascoste* in una applicazione per poter fare delle operazioni che normalmente non si avrebbe l'autorizzazioni di fare.

→ è un accesso indiretto; è *il processo che Jane sta eseguendo* a chiedere l'accesso ai file

Capitolo 3

Mandatory (MAC) policies

Partono dall'assunzione che c'è una differenza tra *utente* e *soggetto*; è ciò che serve per bloccare i *trojan horses*:

- **Utente:** essere umano (di cui mi fido)
- **Soggetto:** processo nel sistema; opera per conto dell'utente; **non sono fidati**

La politica più comune è quella **multilivello**: ogni soggetto e oggetto sono classificate con una etichetta. Si differenziano in politiche che si focalizzano su:

- confidenzialità (Bell La Padula)

  oppure

- integrità (Biba)

3.1 Classificazione di sicurezza

Ogni soggetto ed oggetto è associato ad una coppia di elementi:

- **Livello di sicurezza:** livelli su cui è definita una relazione d'ordine totale (li posso mettere *in fila*).

  $$Secret > Confidential > Unclassified$$

- **Categoria:** insieme di elementi su cui non è definita alcuna relazione di ordinamento; serve per partizionare aree differenti del sistema. Ad esempio, l'università ha un sacco di informazioni di vario tipo: anagrafiche, finanziare, accademiche, ...

  Hanno l'obiettivo di classificarle in classi diverse. Viene fatta sia lato oggetto che lato soggetto.

La combinazione di queste due permette di definire una **relazione di dominanza:**

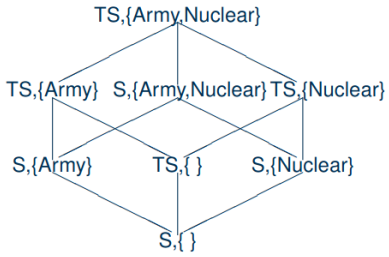
$$(L_1, C_1) \geq (L_2, C_2) \Leftrightarrow L_1 \geq L_2 \wedge C_1 \supseteq C_2$$

Questa relazione soddisfa una serie di proprietà che, in matematica, permette di formare un *reticolo* (quando combinata fra tutte le classi); nel nostro caso parliamo di **reticolo di classificazione**.

- Reflexivity of $\succeq$ $\quad \forall x \in SC : x \succeq x$
- Transitivity of $\succeq$ $\quad \forall x, y, z \in SC : x \succeq y, y \succeq z \Longrightarrow x \succeq z$
- Antisymmetry of $\succeq$ $\quad \forall x, y \in SC : x \succeq y, y \succeq x \Longrightarrow x = y$
- Least upper bound $\quad \forall x, y \in SC : \exists\, !z \in SC$
  - $z \succeq x$ and $z \succeq y$
  - $\forall t \in SC : t \succeq x$ and $t \succeq y \Longrightarrow t \succeq z$.
- Greatest lower bound $\quad \forall x, y \in SC : \exists\, !z \in SC$
  - $x \succeq z$ and $y \succeq z$
  - $\forall t \in SC : x \succeq t$ and $y \succeq t \Longrightarrow z \succeq t$.

Esempio di reticolo di classificazione:

Levels: Top Secret (TS), Secret (S)
Categories: Army, Nuclear



- lub(⟨TS,{Nuclear}⟩,⟨S,{Army,Nuclear}⟩) = ⟨TS,{Army,Nuclear}⟩
- glb(⟨TS,{Nuclear}⟩,⟨S,{Army,Nuclear}⟩) = ⟨S,{Nuclear}⟩

3.2 Semantica della classificazione di sicurezza

- **Classi di sicurezza**
  - associato ad un *soggetto*, rilfette la fiducia verso quell'utente; quanto mi fido di quell'utente
  - associato ad un *oggetto*, riflette la sensisibilità dell'informazione
- Le **categorie** definiscono l'area di competenza di utenti e dati.

3.3 Bell La Padula

È un modello che si preoccupa della confidenzialità (e non del resto).
Considerando di essere in un ambiente multilivello, l'**obiettivo** è prevenire flussi di informazioni ai livelli più bassi o a classi incomparabili.

- ***Simple property:*** un soggetto s può leggere un oggetto o solo se $\lambda(s) \geq \lambda(o)$
- ****-property:*** un soggetto s può scrivere un oggetto o solo se $\lambda(o) \geq \lambda(s)$

$\Rightarrow$ **NO READ UP**
$\Rightarrow$ **NO WRITE DOWN**
Se sono Secret, e scrivo un file secret in uno Top-Secret, non è mica un problema per la confidenzialità. Potrebbe esserlo se scrivo secret in un file Unclassified (potrebbe causare problemi a livelli di integrità, ma ci stiamo occupando solo di confidenzialità).

3.3.1 Proprietà di sicurezza

Un sistema viene formalizzato come *stato* e *transizioni di stato*.
Uno stato $v \in V$ è una tripla (b, M, λ), dove:

- b è l'insieme di triple $(S \times O \times A)$, ovvero l'insieme degli accessi (*soggetto, oggetto, accesso*)
- M è la matrice di accesso (per rappresentare una politica discrezionale)
- λ è una funzione che ritorna la classe di sicurezza associata a soggetti e oggetti

Da questa definizione segue che uno stato (b, M, λ) è sicuro se rispetta:

- **Simple security:** $\forall(s, o, a) \in b, a = read \Rightarrow \lambda(s) \geq \lambda(o)$

  se la tripla appartiene a b, è perché vale la simple property (classe del soggetto domina quella dell'oggetto)

- *** property:** $\forall(s, o, a) \in b, a = write \Rightarrow \lambda(o) \geq \lambda(s)$

  *se la tripla appartiene a b, è perché vale la * property (classe del oggetto domina quella del soggetto)*

Dato che un sistema può variare nel tempo, occorre formalizzare anche questo aspetto. Viene definita una **funzione di transizione di stato** $T : V \times R \to V$, che trasforma lo stato in un altro che soddisfa le due proprietà.

Sistema sicuro

Un sistema (v_0, R, T) è sicuro se ogni stato raggiungibile da v_0 con una sequenza finita di passi da R è sicuro. Formalmente, un sistema è sicuro se:

- v_0 è sicuro
- T è tale che $\forall v$ raggiungibile da v_0 eseguendo una o più richieste da R, allora deve valere che:
  - $(s, o, \texttt{read}) \in b' \land (s, o, \texttt{read}) \notin b \Rightarrow \lambda'(s) \succeq \lambda'(o)$
  - $(s, o, \texttt{read}) \in b \land \lambda'(s) \not\succeq \lambda'(o) \Rightarrow (s, o, \texttt{read}) \notin b'$
  - $(s, o, \texttt{write}) \in b' \land (s, o, \texttt{write}) \notin b \Rightarrow \lambda'(o) \succeq \lambda'(s)$
  - $(s, o, \texttt{write}) \in b \land \lambda'(o) \not\succeq \lambda'(s) \Rightarrow (s, o, \texttt{write}) \notin b'$

3.3.2 *BLP + tranquility*

L'idea è di controllare ciò che fa T, dato che potrebbe dar vita a problemi di sicurezza.
Viene introdotta la ***tranquility property***, secondo cui il livello di sicurezza associato a soggetti e oggetti non può cambiare.
Tuttavia, questa restrizione è troppo forte da poter applicare in casi reali; vengono così introdotti alcuni casi in cui viene *attenuata*:

- non tutti i cambi di livello rilasciano informazioni (cambiare verso l'alto può essere ok)
- soggetti fidati possono fare downgrade

Altre eccezioni

Altre eccezzioni non catturate dal modello, che richiedono di attenuare le restrizioni:

- **Data association:** un set di valori associati potrebbe essere classificato *più alto* rispetto ai valori presi singolarmente
- **Aggregation:** l'aggregazione di tutte le istanze diventa sensibili (ad esempio, la posizione di navi da guerra)
- **Sanitizzazione e downgrading:** dopo un certo tempo alcuni dati potrebbero subire un downgrade

3.3.3 Coesistenza di DAC e MAC

Il modello BLP permette la coesistenza di DAC e MAC: oltre alle classi e alla relazione di dominanza, viene controllata anche la matrice di accesso.

$$\text{DAC property: } b \subseteq \{(s, o, a) | a \in M[s, o]\}$$

Se sono applicati sia DAC che MAC, solo gli accessi che li soddisfano entrambi sono garantiti.

3.3.4 Limitazioni delle politiche mandatorie

Le politiche mandatorie sono vulnerabili ai **canali nascosti**; sono canali normalmente non usati per la comunicazione, ma che possono essere usati per comunicare in modo illegittimo delle informazioni.
Ogni risorsa del sistema condivisa tra processi di livelli diversi può essere usata per creare un canale nascosto. I canali nascosti possono essere di tipo:

- **Storage:** ad esempio un soggetto top-secret crea un file se vuole comunicare qualcosa, altrimenti no; un soggetto secret prova a scrivere tale file: dall'esistenza o meno deduce l'informazione
- **Timing:** viene sfruttata la diversa reazione che il sistema può avere in termini di tempo di risposta.

  Ad esempio, blocco una stampante così che l'utente al livello più basso, vedendo il tempo di risposta diverso, capisce che l'altro utente vuole comunicare qualcosa

3.4 Politche mandatorie per l'integrità

L'altra proprietà da proteggere (oltre alla confidenzialità) è l'integrità. In maniera simile a quanto visto precedentemente, vengono definite:

- **Classi di integrità:** di integrità:
  - associate agli *utenti* riflettono la fiducia che non modifichi informazioni in modo improprio (prima era che non vada a diffondere informazioni sensibili)
  - associate agli *oggetti* riflettono il grado di fiducia che ho dell'informazione contenuta nell'oggetto e il danno potenziale che modifiche a tale oggetto potrebbero causare
- **Categorie:** defininoscono l'area di competenza di soggetti e oggetti

3.5 Modello Biba

È simmetrico a BLP:

- in BLP, non posso leggere verso l'alto e non posso scrivere verso il basso perché non mi fido dei processi (*chi mi dice che un processo non legge informazioni TS e le scrive in oggetti S?*)
- in Biba, i soggetti non possono scrivere a livelli più alti, altrimenti comprometterebbero l'integrità; non posso leggere a livello più basso perché potrei avere informazioni non affidabili

→ vengono fatti ragionamenti opposti perché si hanno obiettivi diversi da raggiungere
La politica di integrità prevede:

- **Simple property:** un soggetto s può leggere un oggetto o solo se $\lambda(o) \geq \lambda(s)$
- ***-property:** un soggetto s può scrivere un oggetto o solo se $\lambda(s) \geq \lambda(o)$

  NB: viene segnalato al sistema ma non bloccato

⇒ **NO READ DOWN**
⇒ **NO WRITE UP**

3.5.1 Politiche alternative

Due varianti per rilassare le restrizioni:

- ***Rilassare l'operazione lettura:***

  Un soggetto s può leggere quasiasi oggetto o. Dopo l'accesso, $\lambda(s) := glb(\lambda(s), \lambda(o))$.

  - *se leggo un informazione a livello più basso, anche le mie azioni si basano su informazioni meno affidabili; per questo viene abbassato il livello di integrità al soggetto (nel caso in cui $\lambda(o) < \lambda(s)$)*
  - *Drawback:* l'ordine delle operazioni influenza i privilegi del soggetto

- ***Rilassare l'operazione di scrittura:***

  Un soggetto s può scrivere qualsiasi oggetto o. Dopo l'accesso, $\lambda(o) := glb(\lambda(s), \lambda(o))$

  - *posso cambiare informazioni affidabili con altre meno affidabili; per questo devo cambiare la classe di integrità dell'oggetto*
  - *Drawback:* non blocca questo tipo di scritture ma si limita a segnarle

3.5.2 Limitazioni di Biba

Il vero problema di Biba è che guarda solo un aspetto limitato dell'integrità delle informazioni; non basta per proteggere l'integrità.

3.6 Applicazione di BLP nelle basi di dati

È prima di tutto necessario decidere il livello di granularità a cui associare le classi di sicurezza per applicare la politica mandatoria:

- relazione
- attributo
- tupla
- elemento

3.6.1 Modello relazionale classico

Ogni relazione è caratterizzata da:

- **Schema** della relazione $R(A_1, \ldots, A_n)$, indipendente dallo stato
- **Istanza** della relazione, dipendente dallo stato, composta da tuple $(a_1, \ldots, a_n)$

Name	Dept	Salary
Bob	Dept1	100K
Ann	Dept2	200K
Sam	Dept1	150K

Gli **attributi chiave** identificano univocamente le tuple:

- due tuple non possono avere la stessa chiave
- le chiavi non possono avere valore `null`

3.6.2 Multilevel DBMSs

Ogni relazione è caratterizzata da:

- **Schema** delle relazione $R(A_1, C_1, \ldots, A_n, C_n)$, indipendente dallo stato
  - C_i, con $i = 1, \ldots, n$, è il range delle classificazioni di sicurezza

  $\rightarrow$ vengono aggiunti altri n attributi per tenere traccia delle classificazioni
- **Set di instanze** della relazione R_c, dipendente dallo stato; una istanza per ogni classe c. Ciascuna istanza è composta da tuple $(a_1, c_1, \ldots, a_n, c_n)$

  L'istanza a livello c contiene solo gli elementi la cui classificazione è dominata da c

Controllo degli accessi

Il controllo degli accessi viene fatto applicando il modello BLP:

- *no read up*
- *no write down further restricted*
  → ogni soggetto può scrivere solo al **suo** livello

Name	λ_N	Dept	λ_D	Salary	λ_S
Bob	U	Dept1	U	100K	U
Ann	S	Dept2	S	200K	S
Sam	U	Dept1	U	150K	S

Instance U

Name	λ_N	Dept	λ_D	Salary	λ_S
Bob	U	Dept1	U	100K	U
Sam	U	Dept1	U	-	U

S-instance is the whole relation

Modello relazionale multilivello

Per ogni tupla in una relazione multilivello:

- gli attributi chiave devono avere la **stessa classe**, altrimenti potrebbero avere valore nullo (riesco ad inferire qualcosa)
- gli **attributi non chiave devono dominare gli attributi chiave**, altrimenti potrei vedere un attributo senza l'identificatore

3.7 Polinstanziazione

La polinstanziazione consiste nella **presenza di oggetti con lo stesso nome ma classifiazione diversa**; ovvero, tuple diverse con la stessa chiave ma:

- diversa classificazione per la chiave (*polyinstantiated tuples*)
- valori e classificazione diversi per uno o più attributi (*polyinstantiated elements*)

Il sistema si occuperà di fare il *merge* tra le varie tuple con classificazione diversa, restituendo sempre una sola tupla.

3.7.1 Polinstanziazione invisibile

Un soggetto a *livello basso* inserisce dati in un campo che contiene già valori a livello più alto; il soggetto non vede i dati già esistenti.

Esempio - Tuple polinstanziate

Name	λ_N	Dept	λ_D	Salary	λ_S
Bob	U	Dept1	U	100K	U
Ann	S	Dept2	S	200K	S
Sam	U	Dept1	U	150K	S

Request by U-subject
INSERT INTO Employee VALUES Ann,Dept1,100K

Name	λ_N	Dept	λ_D	Salary	λ_S
Bob	U	Dept1	U	100K	U
Ann	S	Dept2	S	200K	S
Sam	U	Dept1	U	150K	S
Ann	U	Dept1	U	100K	U

- Un utente U vuole inserire una tupla per Ann, dato che quella già esistente non la vede
- L'inserimento di questa tupla viene permesso, dunque mi ritrovo con due tuple con lo **stesso valore per l'attributo chiave** (Ann), ma con **classificazione diversa**
  - non può essere impedito, perché altrimenti inferisci che Ann c'è ma non la vedi (***information leakage***)
  - non può essere modificata la tupla già esistente (***loss of integrity***)

Esempio - Elementi polinstanziati

Name	λ_N	Dept	λ_D	Salary	λ_S
Bob	U	Dept1	U	100K	U
Ann	S	Dept2	S	200K	S
Sam	U	Dept1	U	150K	S

Request by U-subject
UPDATE Employee SET Salary="100K" WHERE Name="Sam"

Name	λ_N	Dept	λ_D	Salary	λ_S
Bob	U	Dept1	U	100K	U
Ann	S	Dept2	S	200K	S
Sam	U	Dept1	U	150K	S
Sam	U	Dept1	U	100K	U

- Un utente U vuole fare update del salario di Sam (dato che non lo può vedere)
- Viene fatta un'altra tupla, dunque mi ritrovo con due tuple con **gli stessi valori per l'attributo chiave, ma valori diversi per attributi non chiave**
  - l'operazione non può essere impedita (***information leakage***)
  - viene creata un'altra tupla perché la modifica viene fatta con una classe minore, quindi la nuova informazione è meno affidabile di quello che ho già (***loss of integrity***)

3.7.2 Polinstanziazione visibile

Un soggetto ad *livello alto* inserisce dati in un campo che contiene già valori a livello più basso; il soggetto vede i dati già esistenti.

Esempio - Tuple polinstanziate

Name	λ_N	Dept	λ_D	Salary	λ_S
Bob	U	Dept1	U	100K	U
Ann	U	Dept1	U	100K	U
Sam	U	Dept1	U	150K	S

Request by S-subject
INSERT INTO Employee VALUES Ann,Dept2,200K

Name	λ_N	Dept	λ_D	Salary	λ_S
Bob	U	Dept1	U	100K	U
Ann	U	Dept1	U	100K	U
Sam	U	Dept1	U	150K	S
Ann	S	Dept2	S	200K	S

- Un utente S vuole modificare una tupla U
- Viene creata una nuova tupla, così che la tupla U non viene toccata (senza quindi creare un canale di inferenza) e salvando la nuova informazione
  - L'operazione non può essere impedita, dato l'informazione S è più affidabile di quella U (***denial of service***)
  - La tupla esistente non può essere direttamente modificata, altrimenti gli utenti U smetterebbero di vederla e potrebbero inferire informazioni (dato che nell'update si andrebbe a cambiare anche la classe di classifiazione d a U a S) (***information leakage***)

Esempio - Elementi polinstanziati

Name	λ_N	Dept	λ_D	Salary	λ_S
Bob	U	Dept1	U	100K	U
Ann	S	Dept2	S	200K	S
Sam	U	Dept1	U	100K	U

Request by S-subject
UPDATE Employee SET Salary="150K" WHERE Name="Sam"

Name	λ_N	Dept	λ_D	Salary	λ_S
Bob	U	Dept1	U	100K	U
Ann	S	Dept2	S	200K	S
Sam	U	Dept1	U	100K	U
Sam	U	Dept1	U	150K	S

- Un utente S vuole modificare il salario di Sam che è U

- Viene fatta una nuova tupla, dove cambia il **valore e la classe di classificazione**, da U a S
  - come prima, non posso bloccare la richiesta (***denial of service***)
  - come prima, non posso modificare direttamente la tupla esistente (***information leakage***)

Esempio di domanda all'esame

Parlami della polinstanziazione.
La polinstanziazione può essere a livello di tupla o di elemento, ed ognuna di queste può essere visibile o non visibile...

3.7.3 Considerazioni

La polinstanziazione viene fatta perché è l'unico modo per fare ciò che viene richiesto senza creare inferenza.
Tuttavia, è necessario usarla perché si è cercato di applicare la politica ad un livello di granularità molto fine (elemento):

- ho il vantaggio di avere più controllo
- ho lo svantaggio di dover ricorrere alla polinstanziazione

Tuttavia, gli svantaggi pesano più degli svantaggi, per cui tipicamente viene applicata a livello di tupla e non di elemento.

Uso di `restricted`

Un'alternativa è quella di usare `restricted` al posto di `null` quando un utente non può vedere un valore; altrimenti solo con `null` non capisco se è perché non esiste o perché non lo posso vedere.

Cover story

L'aspetto positivo della polinstanziazione sono le cover story, ovvero *qualcosa che ti faccio vedere come vero per coprire la verità.*

3.7.4 Multilevel DBMS - Architetture

Possiamo aver due scenari a seconda del livello di fiducia:

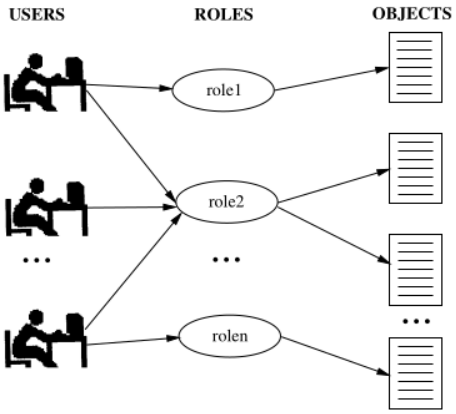
- ***Mi fido di chi gestisce le informazioni:*** vengono memorizzate tutto nello stesso database, perché mi fido che i livelli di sicurezza e la politica mandatoria vengano rispettati
- ***Non mi fido di chi gestisce le informazioni:*** uso diversi database ciascuno dei quali si occupa di gestire un determinato livello di sicurezza, per evitare di mischiare informazioni di livelli diversi; ci deve comunque essere fiducia a livello operativo, che si occuperà di gestire le richieste

Capitolo 4

RBAC policies

Le politiche basate su ruolo sono innovative rispetto a quelle tradizionali come DAC e MAC.
Si passa dall'avere una tripla (s, o, a) in cui si ha un singolo soggetto ad avere un gruppo di soggetti; viene così introdotto il concetto di **ruolo**, ovvero un insieme di privilegi.

- attivando un ruolo r, ad un utente sono concessi tutti gli accessi che sono concessi a r
- l'idea è di avere delle autorizzazioni non sulla base della propria identità, ma sulla base del ruolo che si ricopre



Il modello è caratterizzato da tre entità:

- utenti, ovvero le persone fisiche che possono attivare un ruolo
- ruoli, ovvero un insieme di privilegi

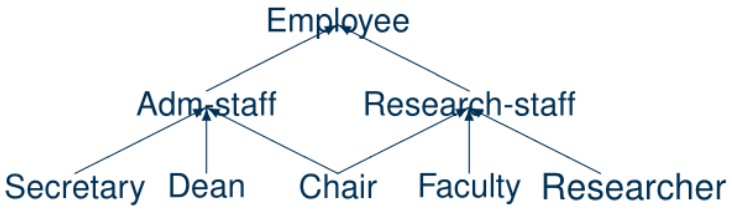
- oggetti, ovvero le risorse a cui è possibile accedere

Nota bene:

- *gruppo* → insieme di *utenti*
- *ruolo* → insieme di *privilegi*

4.1 Gerarchia di specializzazione dei ruoli

È possibile definire una gerarchia di specializzazione sui ruoli.



Questa induce una propagazione di autorizzazioni:

- se ad un ruolo r viene dato un privilegio, automaticamente viene dato anche a tutte le specializzazioni di r
- se un utente u è autorizzato ad attivare un ruolo r, automaticamente è autorizzato ad attivare anche tutti i ruoli che sono una generalizzazione di r

4.2 Vantaggi

- La **specifica delle autorizzazioni è semplificata**
- La gerarchia di ruoli fa in modo di avere delle **autorizzazioni implicite**
- È possibile specificare **restrizioni sui ruoli** che un utente può attivare, come cardinalità o mutua esclusione
- Si definisce un ruolo per ogni *attività nell'organizzazione*, in modo che ciascun utente abbia il **privilegio minimo**, ovvero solo i privilegi strettamente necessari
- Permette la **separazione dei compiti**

4.3 Altri modelli basati su ruoli

Ci sono altri aspetti con l'obiettivo di estendere ancora di più il potere esprissivo di questo modello:

- vincoli sulla propagazione dei ruoli
- fare riferimento anche all'identità del soggetto; si mischia il concetto di ruolo con quello di identità

  La mia segretaria può leggere i miei file

- separazione dei compiti dinamica

Capitolo 5

Politche amministrative

È necessario avere una politica amministrativa che stabilisca chi può definire e gestire le autorizzazioni del sistema; ci sono due scenari:

- **Centralizzata:** ho un unico soggetto che definisce le autorizzazioni; tipico delle politiche mandatorie
- **Ownership:** chi crea le risorse nel sistema ne diventa proprietario, e può definire su esse le autorizzazioni; a differenza del caso precedente l'amminstrazione è *distribuita*.

5.1 Amministrazione decentralizzata

L'amministrazione decentralizzata deve rispondere in modo non ambiguo ad una serie di domande:

- Si possono delegare autorizzazioni? In modo libero o con restrizioni?
- Chi può revocare autorizzazioni?
- Cosa succede alle autorizzazioni concesse da un utente a cui ho rimosso l'autorizzazione di delegare autorizzazioni?

In base a come si risponde a queste domande si otterranno delle politiche diverse; da una parte sono potenti, dall'altro lato è difficile gestire tutti questi aspetti.

5.2 Esempio con SQL

In SQL è possibile creare politiche decentralizzate basate sul concetto di *ownership*:

- è possibile concedere la possibilità di delegare una autorizzazione con `grant-option`

- si crea in questo modo una *catena di autorizzazioni*
- le autorizzazioni possono essere revocate solo da chi le ha concesse

Ann

```
GRANT SELECT
ON TABLE Department
WITH GRANT OPTION
TO Bob
```

Ann

```
GRANT SELECT
ON TABLE Department
TO Chris
```

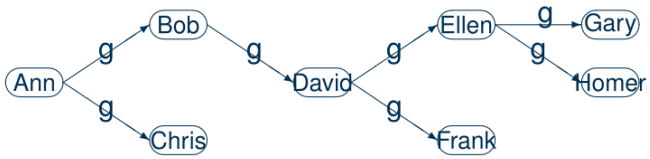
5.2.1 Revoca delle amministrazioni

Se revoco il privilegio a qualcuno che a sua volta lo ha dato a qualcun'altro, quello concesso da questo utente deve essere tolto a sua volta o deve rimanere? Ci sono due modi diversi di operare:

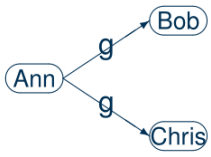
- **with cascade:** la revoca è ricorsiva; bisogna fare attenzioni ai cicli, devo sempre avere un grafo connesso
- **without cascade:** significa che se voglio togliere un privilegio senza il quale un altro privilegio non potrebbe esistere, allora non è possibile fare l'operazione di revoca (prima dovrebbe essere revocato l'altro privilegio)

Esempio

Ogni arco rappresenta una autorizzazione alla stessa risorsa con la stessa modalità di accesso.

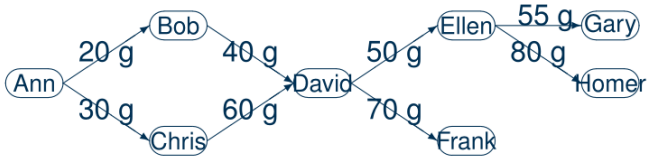


Bob revokes the authorization from David with cascade: result

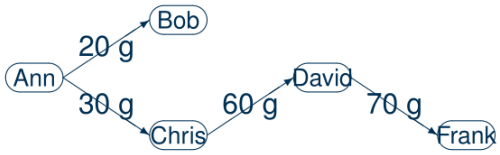


Esempio con tempo

Originariamente su SQL la revoca era basata sul concetto di tempo (ora non più).



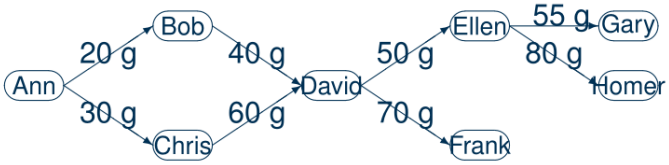
Bob revokes the authorization from David with cascade: result



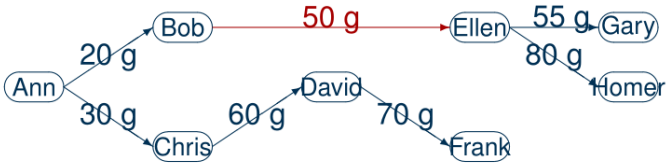
In questo caso, Bob può dare l'autorizzazione a Frank perché l'istante 70 viene dopo l'istante 40 (quello della revoca).

5.2.2 Possibile estensione

Non è detto che si voglia sempre avere una revoca ricorsiva; una possibile alternativa prevede di specificare chi fa la revoca come colui che dà l'autorizzazione (che viene tolta).



Bob revokes the authorization from David: step 2

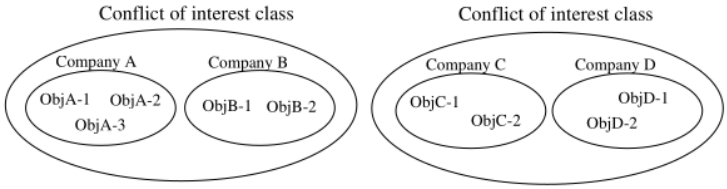


Capitolo 6

Chinese Wall

È il primo modello che ha introdotto il concetto di **separazione dei beni** per proteggere la segretezza; l'obiettivo è prevenire flussi di informazione tra entità che sono in conflitto di interesse.
Gli oggetti sono organizzati gerarchicamente in tre livelli:

- **oggetti base** che contengono informazioni da proteggere
- **company dataset**, ciascuna con i suoi oggetti base
- **classi di conflitto di interesse**



6.1 Proprietà

6.1.1 Simple security rule

Un soggetto s può **accedere** ad un oggetto o solo se:

- o è nello stesso company dataset degli altri oggetti a cui s ha già acceduto
- o appartiene ad una classe di conflitto di interesse diversa da quelle a cui s ha già acceduto

6.1.2 *-property

Un soggetto s può **scrivere** un oggetto o solo se:

- l'accesso è consenito dalla simple security rule, e
- nessun oggetto può essere letto da s (secondo le autorizzazioni) che:
  - è in un company dataset diverso da quello di o, e
  - contiene informazioni *non sanitizzate*

Questa proprietà previene la collusione tra utenti.

6.1.3 Considerazioni

Nella pratica non è usato, si potrebbe addirittura arrivare al punto in cui il sistema diventa inutilizzabile; inoltre, bisogna mantenere uno storico degli accessi.
Tuttavia, è importante per l'introduzione del concetto di separazione dinamica dei compiti.

6.2 Separazione dei compiti

Il principio è quello di dividere le autorizzazioni in modo che nessun utente abbia **troppo potere** in modo da poter abusare del sistema. Può essere fatta in due modi:

- **Statica:** vengono definite in modo esplicito tutte le autorizzazioni
- **Dinamico:** di base tutti possono fare tutto, ma per determinate operazioni viene richiesto che almeno X utenti siano coinvolti

Capitolo 7

Espansione delle autorizzazioni nelle politiche discrezionali

Alcune possibili espansioni possono essere:

- non definire le politiche sui singoli soggetti, ma usare **gruppi di utenti**: tutti i soggetti che appartengono ad un gruppo hanno il privilegio. Le autorizzazioni sono definite a livello di gruppo, non più sull'identità del soggetto.

  Il concetto di gruppo può essere esteso anche ad oggetti e modalità di accesso.

- Si introducono delle **condizioni** da soddisfare affinché l'autorizzazione sia valida; ad esempio:

  - posso accedere solo dalla rete interna
  - posso accedere solo dalle 8 alle 20
  - ...

Autorizzazioni negative

L'utilità di questa astrazione è limitata se non è possibile esprimere delle **eccezioni**. Per questo motivo si possono esprimere anche **autorizzazioni negative**. Ad esempio:

- `(Employees, read, file, +)`
- `(Sam, read, file, -)`

Questo può indurre ad avere **inconsistenza** nella politica di autorizzazione. Come dovrebbe gestirla il sistema?

7.1 Permessi e negazioni

- **Politica aperta:** tutti possono fare tutto, a meno che sia vietato in maniera esplicita
- **Politica chiusa:** nessuno può fare niente, a meno che ci sia una autorizzazione esplicita che lo permetta

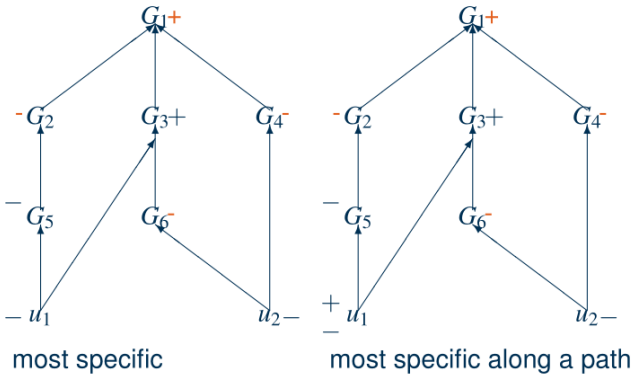
Nella pratica si usanno approcci ibridi, ad esempio una autorizzazione positiva a livello di gruppo ma negativa a livello di utente.
Questo porta con sé due possibili conseguenze:

- **inconsistenza:** per un accesso c'è si + che -
- **incompletezza:** per un accesso non c'è né + né -
  - esigere che per ogni possibile sia specificato qualcosa è *too heavy*; si può risolvere usando una politica di **default**

7.1.1 Politiche per la risoluzione dei conflitti

Alcune possibile politiche possono essere:

- **denials-take-precedence** (vince la negativa)
- **most-specific-takes-precedence** (vince la più specifica)
- **most-specific-along-path-takes-precedence** (tutti i percorsi che vanno verso la radice vanno considerati separatamente)



È possibile definire due classi di autorizzazioni per aiutare la risoluzione dei conflitti (ad esempio nel caso una sia più specifica rispetto al soggetto e una rispetto all'oggetto):

- **strong**: le autorizzazioni forti non possono essere sovrascritte

- **weak**: le autorizzazioni deboli possono essere annullate

Altre politiche di risoluzione dei conflitti:

- Priorità esplicità: le autorizzazioni a priorità esplicita hanno priorità esplicite associate (difficile da gestire)
- La forza posizionale delle autorizzazioni dipende dall'ordine nell'elenco delle autorizzazioni
- La forza delle autorizzazioni dipendente dal concedente dipende da chi le ha concesse
- La forza delle autorizzazioni dipendente dal tempo dipende dal momento in cui sono state concesse (esempio: vincono più recenti)