BD 2 - Sicurezza dei Database

Luca Cosmo

Università Ca' Foscari Venezia

Università
Ca'Foscari
Venezia

Introduzione

Tutti i DBMS principali implementano meccanismi di:

- autenticazione: identificare chi sta operando sul database
- autorizzazione: determinare chi può fare cosa (tramite permessi)

L'autenticazione è normalmente effettuata tramite l'utilizzo di un nome utente ed una password. E' un prerequisito per l'autorizzazione.

Controllo degli Accessi

Il controllo degli accessi è il meccanismo con cui viene verificato che chi richiede un'operazione sia effettivamente autorizzato a farla.

Autenticazione

La gestione degli utenti non fa parte dello standard SQL. Ci focalizziamo sull'implementazione di Postgres, ma altri DBMS sono simili.

Il modo più semplice per creare utenti è tramite la sintassi:

```
CREATE USER NomeUtente WITH PASSWORD NuovaPwd
```

Ulteriori opzioni popolari da aggiungere in coda al comando:

- `SUPERUSER`: l'utente ignora tutti i controlli di sicurezza
- `CREATEDB`: consente la creazione di nuovi database
- `VALID UNTIL ts`: specifica la durata massima della password

Autenticazione

Postgres permette di controllare il processo di autenticazione tramite il file di configurazione `pg_hba.conf`

E' possibile specificare il metodo di autenticazione desiderato per ciascuna richiesta di autenticazione, definita da una quadrupla che include:

1. tipo di connessione: locale, remota, cifrata (via TLS)...
2. database: lista di database o keyword `all`
3. utente: lista di utenti o keyword `all`
4. indirizzo: hostname, indirizzo IP, range di IP...

La prima quadrupla che fa match definisce il metodo di autenticazione. Se non si trova alcuna quadrupla valida, l'autenticazione è vietata.

I metodi di autenticazione supportati includono: trust, reject, password, MD5, SCRAM, peer...

Metodi di Autenticazione: Password

Protocollo per l'utente peter con password 123456:

1. In fase di creazione utente, il server salva $y = 123456$
2. Il client manda lo username (peter) e richiede una connessione
3. Il server richiede la password per lo username peter
4. Il client fornisce la propria password $x = 123456$
5. Il server verifica che $x = y$ ed autorizza l'accesso

Problemi di sicurezza:

- Se il canale di comunicazione non è cifrato, la password è esposta: questo problema è facile da risolvere tramite l'uso di SSL / TLS
- La password è memorizzata in chiaro sul server[1], quindi esposta a chiunque riuscisse ad ottenerne il controllo

[1] Solo per vecchie versioni di Postgres

Metodi di Autenticazione: MD5

Protocollo per l'utente peter con password 123456:

1. In fase di creazione utente, il server salva $y = \text{MD5}(123456 + \text{peter})$
2. Il client manda lo username (peter) e richiede una connessione
3. Il server richiede un MD5 della password, proponendo un salt abcd
4. Il client calcola $x = \text{MD5}(\text{MD5}(123456 + \text{peter}) + \text{abcd})$ e poi lo invia al server
5. Il server verifica che $x = \text{MD5}(y + \text{abcd})$ ed autorizza l'accesso

Questo protocollo non richiede di memorizzare la password in chiaro sul server, ma non è più raccomandato a causa dell'uso di MD5 e del salt corto. Inoltre il furto di y permette l'impersonazione dell'utente!

Per evitare questi problemi è meglio utilizzare il più complesso protocollo SCRAM, che non approfondiremo.

Autorizzazione

Dopo la fase di autenticazione il DBMS sa con chi sta interagendo e può implementare appropriate politiche di autorizzazione.

Regole base:

1. quando un oggetto (es. una tabella) viene creato, il suo creatore ne diventa il proprietario e può farne ciò che desidera
2. gli altri utenti invece possono accedere all'oggetto solamente con le modalità stabilite dai permessi concessi su di esso
3. il privilegio di eliminare un oggetto o alterarne la definizione è di pertinenza esclusiva del creatore dell'oggetto!

Permessi

SQL mette a disposizione diversi tipi di permessi, fra cui:

1. SELECT su una tabella (opz. ristretta ad un set di attributi X)
2. INSERT su una tabella (opz. ristretta ad un set di attributi X)
3. UPDATE su una tabella (opz. ristretta ad un set di attributi X)
4. DELETE su una tabella (SELECT richiesto per DELETE non banali)
5. TRIGGER, necessario per definire un trigger su una tabella
6. EXECUTE, necessario per eseguire una funzione o procedura

Si noti che SELECT e SELECT(X) sono due permessi differenti, dove il primo è più generale del secondo.

Esempio

```
INSERT INTO Studio(name)
  SELECT DISTINCT studioName
  FROM Movies
  WHERE studioName NOT IN
    (SELECT name
     FROM Studio);
```

Questa query ha bisogno di tutti i seguenti permessi:

- `INSERT(name)` su `Studio`
- `SELECT(studioName)` su `Movies`
- `SELECT(name)` su `Studio`

In alternativa si possono avere permessi più generali di questi.

Permessi e Trigger

La gestione dei permessi è delicata per i trigger:

- il permesso TRIGGER per una tabella abilita la definizione di trigger arbitrari su di essa
- il creatore del trigger deve avere il permesso TRIGGER sulla tabella e tutti i permessi richiesti per eseguire l'azione del trigger
- quando un trigger viene attivato, esso viene eseguito con i permessi del suo creatore, indipendentemente da chi ha indotto l'attivazione

Attenzione quindi che l'uso di trigger può abilitare scalate di privilegi!

Permessi e Funzioni

Quando una funzione viene dichiarata, è possibile specificarne i permessi di esecuzione tramite le opzioni:

- `SECURITY INVOKER`: la funzione viene eseguita con i permessi dell'utente chiamante (default)
- `SECURITY DEFINER`: la funzione viene eseguita con i permessi dell'utente che l'ha definita

Sebbene l'uso di `SECURITY DEFINER` possa abilitare scalate di privilegi, un suo utilizzo sapiente può essere utile per fornire accesso controllato a funzionalità che richiedono permessi elevati.

Assegnare Permessi

Il proprietario di uno schema relazionale ha tutti i permessi possibili sulle tabelle e gli altri elementi di tale schema. Tali permessi possono essere concessi ad altri utenti usando la sintassi:

```
GRANT ListaPermessi ON Elemento TO ListaUtenti
```

Alcune note:

- è possibile utilizzare `ALL PRIVILEGES` per indicare tutti i permessi
- è possibile utilizzare `PUBLIC` per autorizzare tutti gli utenti, compresi quelli non ancora esistenti

Delegare Permessi

I permessi possono essere assegnati fornendo la possibilità di delegarli ad altri utenti:

```
GRANT ListaPermessi ON Elemento TO ListaUtenti
WITH GRANT OPTION
```

E' sempre possibile delegare una versione meno generale di un privilegio (delegabile) che si possiede.

Example

Un utente che ha ricevuto il permesso SELECT WITH GRANT OPTION può delegare SELECT(X) con X arbitrario sullo stesso elemento.

Diagramma di Autorizzazione

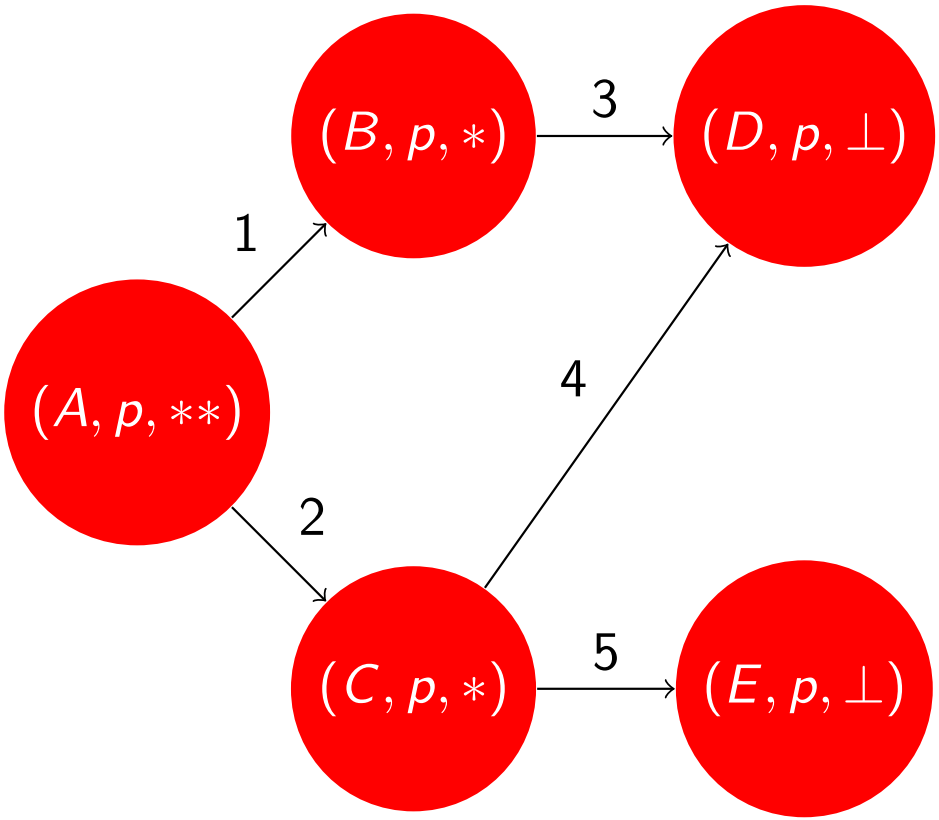
Definition

Un diagramma di autorizzazione è un grafo orientato i cui nodi sono etichettati con una tripla (u, p, m), dove u è un utente, p è un permesso e m può avere una delle seguenti forme:

1. $\perp$: il permesso è stato assegnato, ma non può essere delegato
2. $*$: il permesso è stato assegnato e può essere delegato
3. $**$: il permesso è stato concesso in qualità di proprietario

Un arco da (u_1, p, m_1) a (u_2, p, m_2) modella che u_1 detiene il permesso p con modalità m_1 e lo ha delegato ad u_2 con modalità m_2.

Esempio



Sia A il proprietario di t:

1. A: GRANT p ON t TO B WITH GRANT OPTION
2. A: GRANT p ON t TO C WITH GRANT OPTION
3. B: GRANT p ON t to D
4. C: GRANT p ON t to D
5. C: GRANT p ON t to E

Revoca di Permessi

I permessi assegnati possono essere revocati tramite la sintassi:

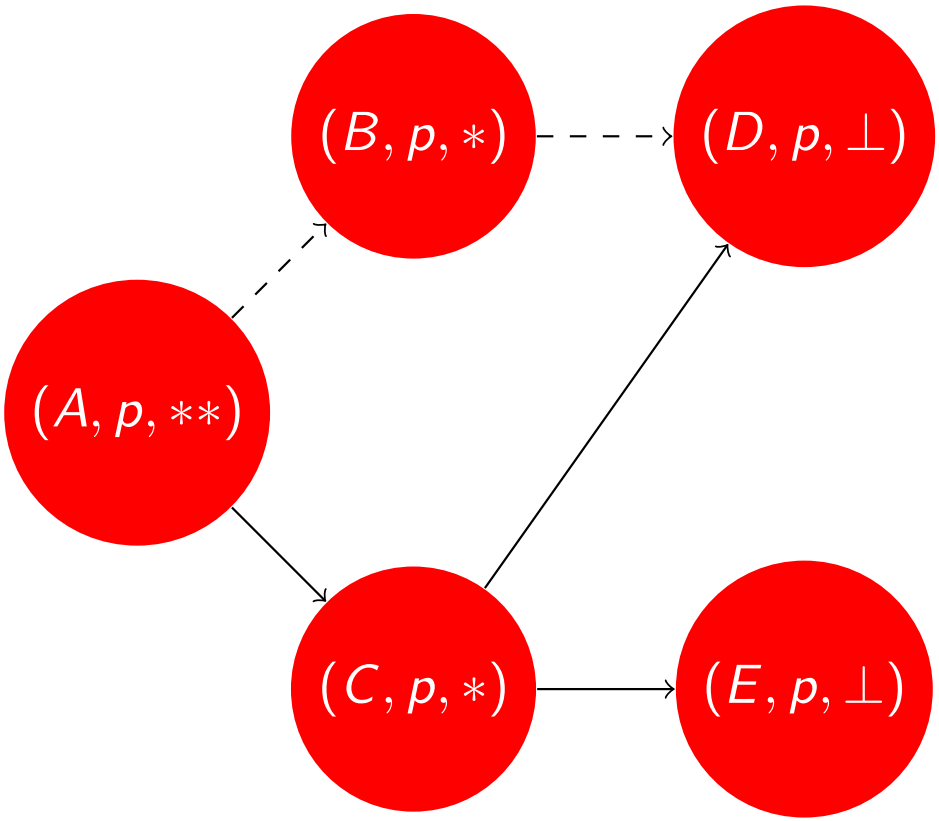
```
REVOKE ListaPermessi ON Elemento FROM ListaUtenti
```

La revoca deve essere terminata da una di queste due opzioni:

1. `CASCADE`: il permesso viene ricorsivamente revocato a tutti gli utenti che lo hanno ricevuto solamente tramite il target della revoca
2. `RESTRICT`: fa fallire la revoca se essa comporterebbe la revoca di ulteriori permessi secondo la politica `CASCADE`

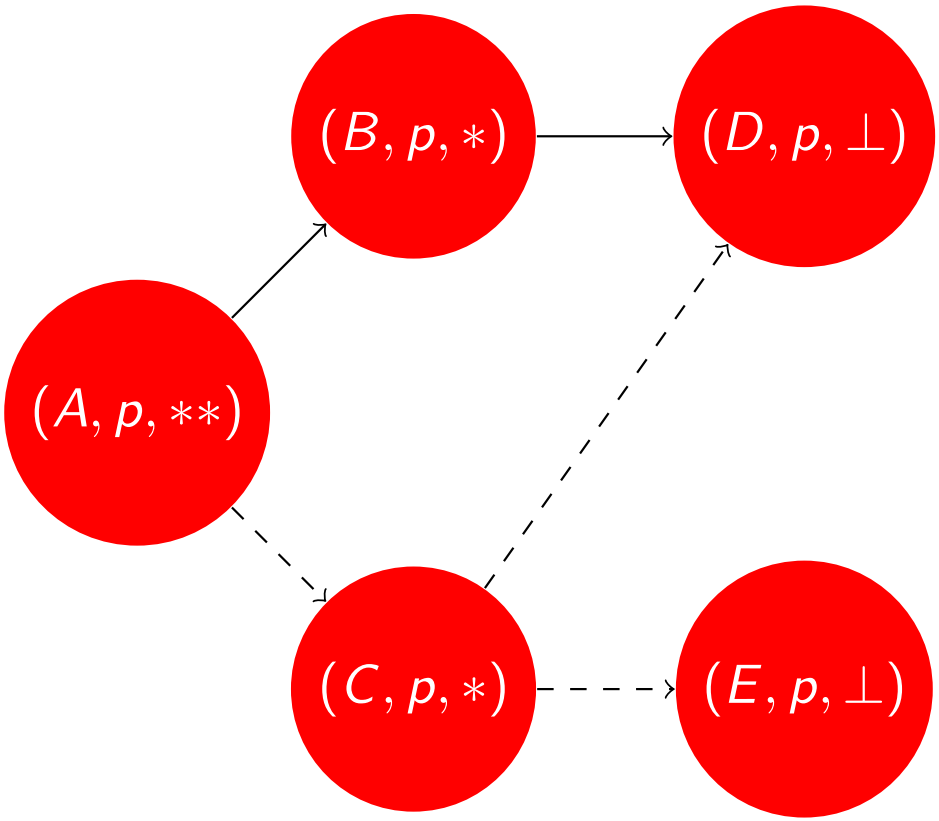
Un utente può revocare soltanto permessi assegnati direttamente da se stesso, a meno di revoche indirette tramite `CASCADE`.

Esempio 1



A: REVOKE p ON t FROM B non revoca il permesso anche a D, dato che esiste ancora un cammino da A verso D

Esempio 2

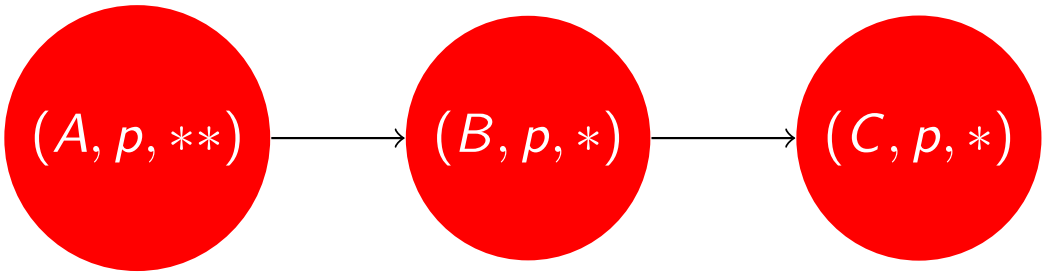


A: REVOKE p ON t FROM C invece revoca il permesso anche ad E, dato che ora non esiste più un cammino da A verso E

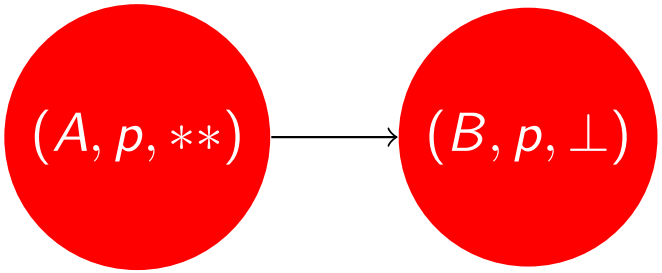
Revoca di Deleghe

E' possibile revocare solo la possibilità di delega, ma non il permesso:

```
REVOKE GRANT OPTION FOR ListaPermessi
                        ON Elemento FROM ListaUtenti
```



Se A revoca a B la possibilità di delegare p (con l'opzione `CASCADE`):



Revoca e Generalità

E' possibile che un utente possieda sia un permesso p che una sua variante meno generale p^- sullo stesso oggetto.

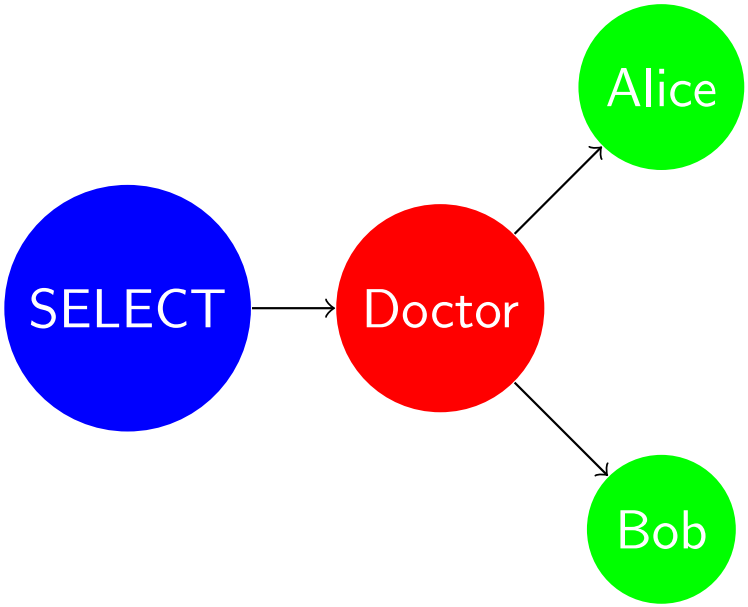
Normalmente revocare p^- non ha alcun effetto su p. Se invece viene revocato p, la scelta dipende dal DBMS:

- Postgres revoca automaticamente anche p^-
- Lo standard SQL invece suggerisce di lasciare assegnato p^-

In generale: l'autorizzazione nei DBMS è sostanzialmente standard, ma ci sono piccole differenze che possono introdurre vulnerabilità inaspettate!

Ruoli

Assegnare manualmente i permessi ad ogni singolo utente è spesso un processo costoso ed error-prone, visto l'elevato numero di utenti.



Un ruolo è un collettore di permessi, che permette di introdurre un livello di indirezione durante la loro assegnazione.

Gestione dei Ruoli

Un ruolo può essere creato tramite il comando:

```
CREATE ROLE NomeRuolo;
```

Una volta fatto ciò è possibile usare alcuni comandi già visti:

- `GRANT`: per assegnare permessi a ruoli e ruoli ad utenti
- `REVOKE`: per rimuovere permessi a ruoli e ruoli ad utenti

I ruoli assegnati ad un utente non sono attivi di default. L'attivazione di un ruolo per ottenerne i permessi viene effettuata tramite il comando:

```
SET ROLE NomeRuolo;
```

Benefici dei Ruoli

I ruoli hanno numerosi benefici rispetto all'uso tradizionale dei permessi:

- i ruoli raggruppano insiemi di permessi logicamente collegati
- è molto meno costoso assegnare ruoli che permessi, visto che ci sono molti meno ruoli che permessi
- è molto più difficile sbagliare l'assegnazione di un ruolo che di un insieme di permessi
- le operazioni di revoca sono analogamente semplificate
- i ruoli non sono attivi di default, contrariamente ai permessi: questo è più fedele al principio del minimo privilegio

Ruoli in Postgres

In Postgres non c'è una vera e propria differenza fra utente e ruolo: infatti il comando `CREATE USER` è un alias per `CREATE ROLE WITH LOGIN`

Alcune specificità:

- l'opzione `CREATEROLE` consente al ruolo di creare altri ruoli. Questo può condurre a scalate di privilegi, da usare con attenzione!
- ruoli assegnati con `WITH ADMIN OPTION` possono essere delegati
- è possibile assegnare ruoli ad altri ruoli, introducendo una forma di ereditarietà dei permessi
- il diagramma di autorizzazione è costruito attorno ai ruoli: se un permesso viene assegnato tramite un ruolo, qualsiasi altro utente con quel ruolo può revocarlo

Ruoli in Postgres: Ereditarietà

Le opzioni `INHERIT` (default) e `NOINHERIT` consentono di gestire il meccanismo di ereditarietà:

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

Che permessi sono concessi?

- login come `joe`: permessi di `joe` e di `admin` (ma non di `wheel`)
- dopo `SET ROLE admin`: solo i permessi di `admin`
- dopo `SET ROLE wheel`: solo i permessi di `wheel`

Sia nel primo che nel secondo caso si può fare `SET ROLE wheel`

Raccomandazioni Generali

Un'opportuna politica di autorizzazione è necessaria per la sicurezza di un database. Alcune raccomandazioni:

1. definite un insieme di ruoli in fase di progettazione del database a partire dall'analisi dei requisiti
2. definite le politiche di confidenzialità ed integrità per il database, mappandole su opportuni assegnamenti di permessi a ruoli
3. assegnate i ruoli agli utenti rispettando il principio del minimo privilegio e facendo attenzione all'utilizzo dell'ereditarietà

Attenzione!

Una buona politica di autorizzazione non è sufficiente per la sicurezza!

SQL Injection

Consideriamo una web application che consenta di cercare le informazioni relative ad un utente, presentando il suo nome ($u) e password ($p).

```
user = get_parameter($u)
pass = get_parameter($p)
statement = "SELECT * FROM users WHERE name = '" +
            user + "' AND pwd = '" + pass + "';"
```

Il codice costruisce la query SQL da eseguire tramite concatenazione di stringhe, che però potrebbero essere state passate da un attaccante...

SQL Injection

```
user = get_parameter($u)
pass = get_parameter($p)
statement = "SELECT * FROM users WHERE name = '" +
              user + "' AND pwd = '" + pass + "';"
```

Se passiamo nome utente `marco` e password `' OR '1' = '1`

```
SELECT * FROM users WHERE name = 'marco' AND pwd = ''
                          OR '1' = '1';
```

Questa query ritorna l'intero contenuto della tabella `users`, andando a comprometterne la confidenzialità!

SQL Injection

```
user = get_parameter($u)
pass = get_parameter($p)
statement = "SELECT * FROM users WHERE name = '" +
            user + "' AND pwd = '" + pwd + "';"
```

Se passiamo nome utente `marco` e password `'; DROP TABLE users --`

```
SELECT * FROM users WHERE name = 'marco' AND pwd = '';
DROP TABLE users -- ';
```

Questa query elimina l'intero contenuto della tabella `users`, andando a comprometterne l'integrità!

Prevenire SQL Injection

Come per tutte le forme di injection, ci sono due approcci tradizionali per prevenire SQL injection:

- sanitizzazione: analisi o trasformazione degli input processati per garantire l'assenza di contenuti malevoli, per esempio costrutti SQL
- encoding: trasformazione degli output generati per garantire che essi non vengano interpretati come codice eseguibile

Nel caso di SQL l'utilizzo dell'encoding è più comune, perché è raro avere casi d'uso in cui del codice SQL deve essere caricato in un'applicazione.

Escaping

L'escaping è una delle più semplici forme di encoding, che converte caratteri speciali nella loro versione "letterale"

```
userName  = escape(get_parameter($u))
pwd       = escape(get_parameter($p))
statement = "SELECT * FROM users WHERE name = '" +
             userName + "' AND password = '" + pwd + "';"
```

Se passiamo nome utente `marco` e password `' OR '1' = '1`

```
SELECT * FROM users WHERE name = 'marco'
                      AND password = ''' OR ''1'' = ''1';
```

Importante: affidatevi a funzioni di libreria esistenti!

Prepared Statement

Un prepared statement è un'istruzione SQL contenente dei "buchi" detti parametri, che vengono riempiti in modo disciplinato (tipato).

```
userName  = get_parameter($u)
pwd       = get_parameter($p)
statement = "SELECT * FROM users WHERE name = ?
                                 AND password = ?;"
statement.setString(1,userName);
statement.setString(2,pwd);
```

I prepared statement sono il modo consigliato per evitare SQL injection, ma in alcuni casi non si possono usare...

Sanitizzazione

Si consideri la seguente porzione di codice usata per stampare il contenuto di una tabella selezionabile da un menù a tendina:

```
table = get_parameter($t)
statement = "SELECT * FROM " + table;
```

Poichè la sintassi dei prepared statement non supporta l'uso di parametri nel nome della tabella, è importante sanitizzare l'input ricevuto.

```
table = get_parameter($t);
if (table == "Student" || table == "Teacher") {
  statement = "SELECT * FROM " + table;
}
else { throw new Exception("Unexpected!"); }
```

Checkpoint

Concetti Chiave

- Autenticazione ed autorizzazione
- Diagramma di autorizzazione: definizione e semantica
- Ruoli: benefici ed implementazione in Postgres
- SQL injection: rischi per la sicurezza e prevenzione

Materiale Didattico

Database Systems: Sezione 10.1. Approfondimenti dal manuale di Postgres (client authentication e database roles).