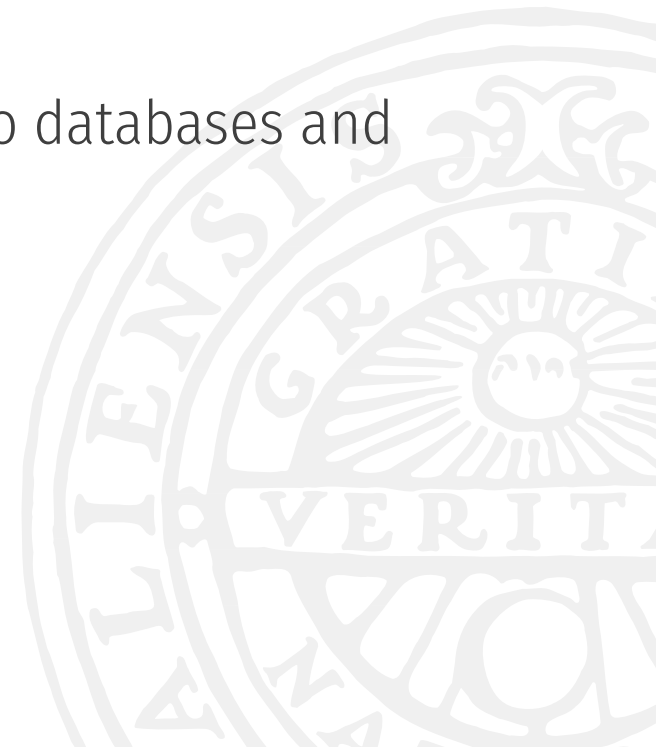# Users, Authorization & Security

1DL301 Database Design I

INTENDED LEARNING OUTCOMES

► Know how to set up users and their privileges in RDBMS

► Understand and be able to prevent SQL injection attacks

► Be able to store passwords in a secure way

► Be aware of other security threats relevant to databases and database servers

CREATING AND DROPPING USERS

Slightly different syntax for different RDBMS.

To create a user in MySQL:

```
CREATE USER user
IDENTIFIED [WITH auth_plugin] BY auth_string
```

Username *user* can be followed by @ and the IP or the hostname from which the user is allowed to log in, e.g., `user@%.se`.

Default is any address except localhost.

Examples:

```
CREATE USER heisenberg@localhost IDENTIFIED BY 'IAmTheOneWhoKnocks!'
CREATE USER heisenberg IDENTIFIED WITH sha256_password BY 'Azul987'
```

To drop a user:

```
DROP USER user
```

SQLite is a single user database system.

Authorization – control of what users can do in the database (what they can see, create, modify, delete etc.)

(Most important) table privileges on $R(A_1, \ldots, A_n)$:

```
SELECT ON R
SELECT(A₁, ..., Aₙ) ON R

INSERT ON R
INSERT(A₁, ..., Aₙ) ON R

UPDATE ON R
UPDATE(A₁, ..., Aₙ) ON R

DELETE ON R
```
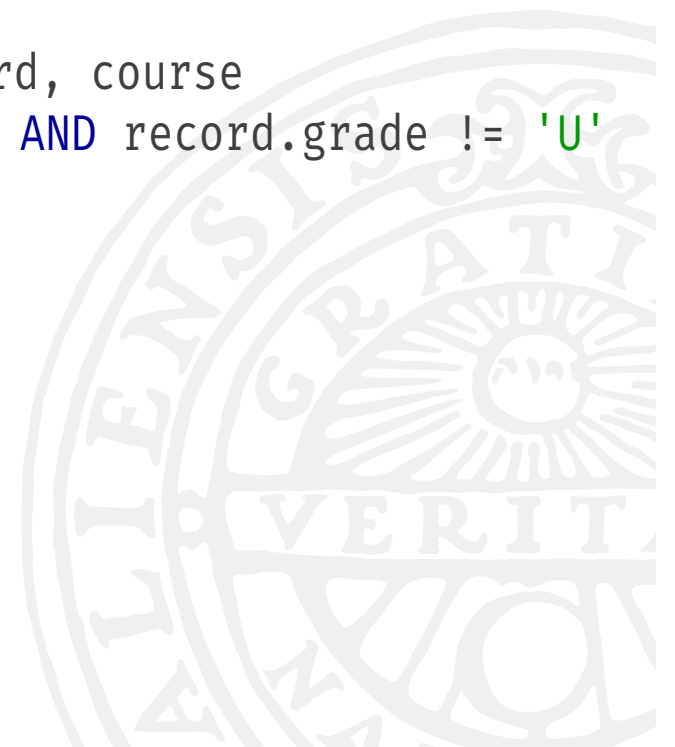
What privileges are needed to be able to execute the following query?
Select all that apply.

```sql
UPDATE student
SET graduated = true
WHERE id IN (SELECT record.student_id FROM record, course
             WHERE record.course_id = course.id AND record.grade != 'U'
             GROUP BY record.student_id
             HAVING SUM(course.credit) >= 180)
```

What privileges are needed to be able to execute the following query? Select all that apply.

```
DELETE FROM project
WHERE id NOT IN (SELECT project_id FROM employee_project)
```

1. SELECT(project_id) ON employee_project

2. SELECT(id) ON project

3. SELECT(project_id) ON project

4. DELETE ON employee_project

5. DELETE(id) ON project

6. DELETE ON project

VIEW PRIVILEGES

What if we want to control the access to certain rows? We can use updatable views.

Privileges for views are exactly the same as for tables.

Example: Each department manager can see, add, modify and delete only his/her own employees.

For the planning department: Grant SELECT, INSERT, UPDATE and DELETE on employees_planning:

```
CREATE VIEW employees_planning AS
SELECT * FROM employees
WHERE department_id = 1
WITH CHECK OPTION
```

The employee_planning view is updatable since we have only used one table and included all its columns (including the primary key). The WITH CHECK OPTION part will ensure that only rows which match the WHERE clause might be inserted into, updated or deleted from the underlying table (employees).

GRANTING PRIVILEGES IN SQL

User who created a table is its owner (and has all privileges) and can grant privileges to other users:

```
GRANT privilege ON object
TO user
[WITH GRANT OPTION]
```

WITH GRANT OPTION – *user* can also grant the privileges (or a subset of them) to other users.

In some RDDBMS you can use PUBLIC as *user* to grant a privilege to all users (both current and future ones).
Note that there are other privileges than those mentioned on the previous slides. Check the reference manual for your RDBMS.

It is possible to grant all privileges using:

```
GRANT ALL ON object TO ...
GRANT ALL ON * TO ...
```

Grant graph – a graph with users as nodes with edges showing which user granted a given privilege to whom.

This graph has an important function when revoking the privileges.

REVOKING PRIVILEGES IN SQL

General syntax:

```
REVOKE privilege ON object
FROM user
[CASCADE | RESTRICT]
```

▶ CASCADE – revoke the previously granted privilege from *user* and from all users which were granted the privilege by a chain of grants started by the *user*

(unless the users were granted the permission by somebody else too)

▶ RESTRICT (default) – do nothing if *user* granted the privilege to other users

MySQL supports neither CASCADE nor RESTRICT, *privilege* is revoked from *user* only.

Alice is the creator of the table Student.
Following SQLs were executed:

```sql
GRANT SELECT ON Student TO Bob; -- executed by Alice
GRANT SELECT ON Student TO Bob; -- executed by Charlie
REVOKE SELECT ON Student FROM Bob; -- executed by Alice
```
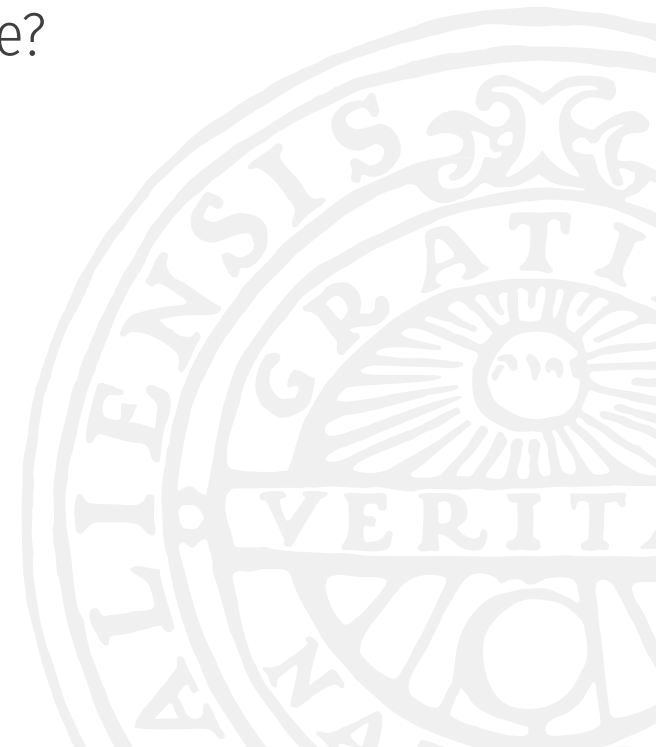
Who has the SELECT privilege on the Student table?

1. Alice and Charlie
2. Alice, Bob and Charlie
3. Alice and Bob
4. Alice
5. Nobody

Bob is the creator of a table called X. Following SQLs were executed:

```
GRANT SELECT ON X TO Jim WITH GRANT OPTION; -- executed by Bob
GRANT SELECT, UPDATE ON X TO Ann WITH GRANT OPTION; -- executed by Bob
GRANT SELECT ON X TO Tim; -- executed by Jim
GRANT SELECT ON X TO Tim; -- executed by Ann
REVOKE SELECT ON X FROM Tim; -- executed by Bob
```

Which privileges does Tim have according to the SQL specification?

1. SELECT with grant option
2. SELECT without grant option
3. SELECT, UPDATE, both with grant option
4. None
5. SELECT with grant option, UPDATE without grant option
6. SELECT, UPDATE, both without grant option

EXERCISE

What does the following program do?
Can you see some problems with the code?

```python
import MySQLdb

conn = MySQLdb.connect("localhost", "root", "pwd", "sqltutorial")
conn.autocommit(True)
cursor = conn.cursor()

while True:
    last_name_prefix = input("Last name starts with: ")
    cursor.execute("SELECT id, first_name, last_name, hour_salary"
        + " FROM employee"
        + " WHERE last_name LIKE '" + last_name_prefix + "%'")
    for row in cursor:
        print("%4d | %32s | %32s | %.2f" % row)

cursor.close()
conn.close()
```

What happens if I enter:

- ► L
- ► `NNN' UNION SELECT id, title, '', 1 FROM department WHERE 1 OR id='`
- ► `NNN' UNION SELECT 0, user, authentication_string, 0 FROM mysql.user WHERE 1 OR user='`
- ► `'; DROP TABLE employee_project; --`
- ► `'; DROP DATABASE sqltutorial; --`

Top 10 most critical web application security risks – 2017 by The Open Web Application Security Project (OWASP):

#1 – Injection

SQL INJECTION

How to prevent SQL injection in your applications?

▶ If you use SQL directly: Always use prepared statements!

```
 9    cursor.execute("SELECT id, first_name, last_name, hour_salary"
10        + " FROM employee"
11        + " WHERE last_name LIKE %s", (last_name_prefix + '%',))
```

▶ Consider using ORM (object-relational mapping)

▶ Disallow multiple statements in your applications
This alone is not enough since it will not stop extending queries with set operations and nested queries.

▶ Use LIMIT (or equivalent) to prevent larger disclosures

When we did the SQL injection to get the users in the MySQL database, one of the rows was:

```
0 | heisenberg | *AE338C5CBDB58198C406C76474CD58F1D0DBA759 | 0.00
```

Go to https://crackstation.net/ and use the hash from the third column:

PASSWORDS, CONT'D

Passwords must never be stored in plain text but as "hashes".

Password hash is calculated using one-way hash functions (i.e., functions which are infeasible to invert).

To check if the password is correct: calculate $hash(password)$ and compare it to the stored hash.

Brute-force attack (if you know the password hash): Try different x and see if $hash(x)$ matches the hash.

There exist precomputed tables with weak passwords and their hashes, like the one which we used on the previous slide.

For more information, google *rainbow tables*.

SALTED PASSWORD HASHES

Prevention

Storing "salted" hashes:

1. Generate random string *salt*
2. $password\_hash = hash(concat(salt, password))$
3. Store both *salt* and *password_hash*

To check if the the password is correct: use the stored *salt*, calculate $hash(concat(salt, password))$ and compare it to the stored hash.

DATABASE RELATED SECURITY THREATS

Top 10 threats related to databases (according to Application Security, Inc.):

- ▶ Default or weak passwords
- ▶ SQL injection
- ▶ Excessive user and group privileges
- ▶ Unnecessary DBMS features enabled
- ▶ Broken configuration management
- ▶ Buffer overflows
- ▶ Privilege escalation
- ▶ Denial of service (DoS)
- ▶ Un-patched RDBMS
- ▶ Unencrypted data

HUMAN FACTORS

▶ Frauds, thefts (e.g., employees stealing backups)

▶ No or insufficient maintenance

▶ No backups

▶ Encryption keys stored on vulnerable places

ADVICES

Writing (server) applications

- ▶ Prepared statements.
- ▶ Avoid connecting to the database as a user with excessive privileges.
- ▶ User inputs must be validated and sanitized on the server side!
- ▶ Use encrypted communication between your application and database server.
- ▶ Use encrypted communication between a client (a browser) and your server (web) application.
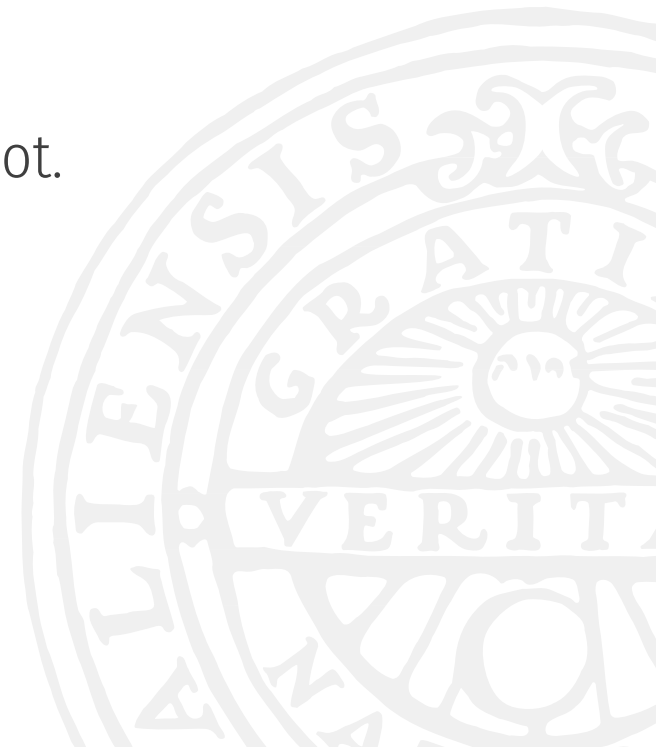- ▶ Store passwords in a safe way.

Network Security

- ► Encryption of network communication.
- ► Run only necessary services.
- ► Set up the firewall.

Operating system security

- ► Don't run database server (e.g., MySQL) as root.
- ► Keep the operating system up-to-date.
- ► Restrict logins on the database host.
- ► Consider encrypting your data.
- ► Back up your system regularly.

JUICE SHOP

If you want to try (and learn how) to exploit security bugs in web applications:

https://github.com/bkimminich/juice-shop

Juice Shop is an intentionally insecure web application for security trainings

It covers all OWASP's Top 10 and other severe security flaws.

QUIZ SOLUTION

Exercise 1: UPDATE(graduated) ON student
　　　　　　　SELECT(id) ON student
　　　　　　　SELECT(student id, course id, grade) ON record
　　　　　　　SELECT(id, credit) ON course

Quiz 1: (1, 2, 6)
Quiz 2: (4)
Quiz 3: (2), according to PostgreSQL doc only users that
　　　　　　gives access can revoke it. MySQL does not
　　　　　　follow standard for revoking privileges.

The slide is derived from
Jan Kudlicka's DB1-Fall19 class.