Database Security

Table of contents

Outline . . . 1
Why database security matters . . . 1
Database control measures . . . 2
Access control mechanisms . . . 3
Discretionary Access Control (DAC) . . . 3
Question 1 — Discretionary access control . . . 6
Question 2 — Discretionary access control (propagation) . . . 6
Mandatory Access Control (MAC) . . . 7
Role-Based Access Control (RBAC) . . . 10
Other control measures . . . 10
SQL injection . . . 11
Summary . . . 13

Module 5 — the final module. Covers threats to database security, the control measures used to defend against them, and SQL injection.

Outline

- Threats to database security
- Access control mechanisms — discretionary, mandatory, role-based
- Other control measures — inference control, flow control, encryption
- SQL injection

Why database security matters

Some of the largest data breaches on record (Yahoo, First American, Facebook, Marriott, MongoDB, Equifax, and others) collectively exposed billions of records — via hacking, poor security practice, or simple misconfiguration. Database security matters for:

- **Legal/policy reasons** — e.g. government policy protecting access to individuals' data (medical records, financial ratings).
- **Ethical reasons** — e.g. controlling who can see employee salary packages or student grades.
- **Technical reasons** — e.g. protection from malware, performance overloading.

Threats — the CIA principle

- **Loss of confidentiality** — unauthorized disclosure of confidential information.
- **Loss of integrity** — improper modification of information.
- **Loss of availability** — a legitimate user cannot access data objects.

Privacy vs. security

- **Security** concerns *how access to data is controlled* — availability for use (with integrity) and permitted access.
- **Privacy** concerns *how data can be used* — an individual's ability to control the terms under which their sensitive data is acquired and used (preventing storage of sensitive data; ensuring appropriate/authorized use of it).
- Security is a **required building block** for privacy, but they're distinct: privacy concerns both directly accessible information *and* information that can be **inferred** from accessible data.

Sensitive data is often double-edged — patient data is private, but also valuable for biomedical/public-health research; audio from CCTV can capture private conversations, but also evidence of criminal activity. This creates a **utility/privacy tradeoff**: protect all sensitive data, while making as much nonsensitive data available as possible.

Database control measures

Four broad categories:

1. **Access control** — provide access only to users with the right access authority.
2. **Inference control** — ensure information about individuals cannot be *derived* even indirectly (applies to statistical databases).
3. **Flow control** — prevent information from flowing to unauthorized users.
4. **Data encryption** — protect sensitive data in transit.

Access control mechanisms

Database access control has two components:

- **Authentication** — verifying the identity of whoever is accessing the database.
- **Authorization** — determining whether an authenticated user should be allowed to execute the transaction they're attempting.

Basic authentication security functions

- **User accounts** — users log in with an assigned username/password.
- **Login session** — a sequence of database operations by one user, recorded in the **system log** (also used for recovery, not just security).
- **Database audit** — reviewing the log to examine all accesses and operations applied during a period, to identify potential privacy breaches or unauthorised modifications. A log typically records who ran what operation, when, and (for modifications) the before/after values.

Three authorization mechanisms

1. **Discretionary Access Control (DAC)** — grant/revoke privileges to users.
2. **Mandatory Access Control (MAC)** — classify data and users into security classes, and implement a security policy across them.
3. **Role-Based Access Control (RBAC)** — assign users to roles, then apply DAC or MAC at the role level.

Discretionary Access Control (DAC)

Based on **granting** and **revoking** privileges.

```
CREATE USER <username> IDENTIFIED BY "<password>";

GRANT privilegeName
ON     objectName
TO     {userName | PUBLIC | roleName}
[WITH GRANT OPTION];

REVOKE privilegeName
ON     objectName
FROM   {userName | PUBLIC | roleName};
```

Two levels of privileges

- **Account level** — privileges specified per account, independent of any particular relation: `CREATE SCHEMA/TABLE/VIEW`, `ALTER/DROP TABLE`, modification (`INSERT/DELETE/UPDATE`) and `SELECT` privileges.
- **Relation level** — privileges for a specific relation or view (can also be specified at the *attribute* level): **select** (read), **modification** (insert/delete/update), and **references** privileges. The `REFERENCES` privilege grants permission to create a *foreign key* reference to the specified table.

Three concrete forms of the above, all of which week12-tutorial-5-1-database-security[1] needs in order to repair a misconfigured set of accounts:

```
-- everything the account holds on that object, in one statement
REVOKE ALL PRIVILEGES ON `schema`.`Table` FROM 'user';

-- the ability to delegate, revoked separately from the privilege itself
REVOKE GRANT OPTION ON `schema`.`Table` FROM 'user';

-- attribute level: the column list goes on the privilege, not the object
GRANT SELECT (`fName`, `lName`) ON `schema`.`Table` TO 'user';
```

Note there is no statement for *narrowing* a privilege: downgrading an account from `ALL PRIVILEGES` to `SELECT` is always `REVOKE ALL PRIVILEGES` followed by `GRANT SELECT`. Backticks quote identifiers that contain spaces or apostrophes, and objects are named `` `schema`.`table` ``.

```
CREATE SCHEMA AUTHORIZATION oe
    CREATE TABLE Product (color VARCHAR2(10) PRIMARY KEY, quantity NUMBER)
    CREATE VIEW RedProduct AS
        SELECT color, quantity FROM Product WHERE color = 'RED'
    GRANT select ON RedProduct TO hr;
```

This single statement creates a schema `oe`, creates `Product`, creates the view `RedProduct`, and grants `hr` the `SELECT` privilege on `RedProduct`.

Access matrix model

Each relation `R` is assigned an **owner** account. Owners can grant select/modification/references privileges to other users on any owned relation. An access matrix `M(i, j)` captures the privilege(s) user `i` has on object `j`:

[1] courses/infs1200/week12-tutorial-5-1-database-security.pdf

	Jake	Aiden	Nelly	Sham
Customer	READ	READ	MODIFY	—
CustOrder	—	—	READ	—
CustOrder.amt	—	READ	READ	—
Supp.status	—	READ	MODIFY	REFERENCE
Supp.address	—	READ	MODIFY	MODIFY

Revoking and propagating privileges

`REVOKE` cancels a privilege — useful for granting something temporarily. If account `A` grants a privilege to `B` **`WITH GRANT OPTION`**, `B` can then grant it onward to other accounts, without `A`'s direct knowledge — the DBMS must track this *chain of dependency* to support revocation correctly.

```
REVOKE privilegeName ON objectName FROM userName [RESTRICT | CASCADE];
```

- **`CASCADE`** — revokes the privilege *and* any dependent privileges that were granted as a result of it (following the chain onward).
- **`RESTRICT`** — refuses the revoke (returns an error) if the privilege has already been passed on to someone else.

A user's authorization is valid **iff there is a path from the root of the authorization graph down to that user's node**. So if `u1` grants to `u2`, who grants to `u3`: revoking `u1 → u2` also cuts off `u3` (no path remains); but revoking a *different* branch, or `u2`'s authorization being revoked by someone else entirely, doesn't necessarily touch `u1` (the grantor's own authorization is independent of what they've granted downstream).

Specifying privileges through views

If owner `A` wants to give account `B` a *limited* capability to `SELECT` from a relation — e.g. only some columns, or only rows meeting a condition — `A` can create a view and grant access to the view instead of the base table:

```
-- Employee [ssn, name, dob, address, sex, salary, mgrSSN, dNum]
CREATE VIEW A3EMPLOYEE AS
    SELECT name, dob, address
    FROM   EMPLOYEE
    WHERE  dNum = 5;

GRANT SELECT ON A3EMPLOYEE TO A3;
```

`A3` can now only ever see `name/dob/address` for department 5 employees — never salary, ssn, or other departments' staff.

Question 1 — Discretionary access control

Which statement removes a privilege from a user?

A. `Remove update on department from Amir` B. `Revoke update on employee from Amir` C. `Delete select on department from Raj` D. `Grant update on employee from Amir`

> **Solution**
>
> **B.** `REVOKE` is the actual SQL keyword for removing a privilege. `REMOVE` and `DELETE` aren't valid DCL keywords for this purpose, and `GRANT` *adds* a privilege rather than removing one.

Question 2 — Discretionary access control (propagation)

`u1` grants authorization to `u2`, who in turn grants it to `u3`. Which is correct?

A. If `u1` revokes from `u3`, `u2`'s authorization is revoked. B. If `u1` revokes from `u2`, `u3`'s authorization is also revoked. C. If `u2`'s authorization is revoked, `u1`'s authorization is also automatically revoked.

> **Solution**
>
> **B.** A user has an authorization iff there's a path from the root of the authorization graph down to their node. `u1 → u2 → u3` — revoking `u1`'s grant to `u2` removes the only path down to `u3` as well, so `u3` loses authorization too. Revoking from `u3` directly (A) has no effect on `u2`'s own (separately-granted) authorization. `u2` losing authorization (C) doesn't propagate *upward* to `u1` — `u1` is the root/grantor, not a recipient.

Worked exercise — GRANT/REVOKE chains

Four users `A1, A2, A3, A4`. DBA: `GRANT CREATETAB TO A1`. `A1` creates `Employee` and `Department`, then runs:

```
GRANT INSERT, DELETE ON Employee, Department TO A2;
GRANT SELECT ON Employee, Department TO A3 WITH GRANT OPTION;
```

Q1: *Can `A3` execute `GRANT SELECT ON Employee TO A4`?*

> **Solution**
>
> **Yes** — `A3` was granted `SELECT WITH GRANT OPTION`, so `A3` can pass that privilege on to `A4`.

Q2: *If `A1` then runs `REVOKE SELECT ON Employee FROM A3`, does `A4` still have `SELECT` on `Employee`?*

> **Solution**
>
> **No** — `A4`'s privilege is automatically revoked too, by propagation (there's no longer a path from the root down to `A4`).

Q3: *What type of access control is this?*

> **Solution**
>
> **Discretionary access control** — privileges are granted/revoked at the *discretion* of individual account owners, rather than being fixed by a system-wide security classification (MAC) or role hierarchy (RBAC).

Mandatory Access Control (MAC)

An additional policy that classifies **both data and users** into security classes, for **multilevel security**. Typical classes (low to high): **Unclassified (U)** < **Confidential (C)** < **Secret (S)** < **Top Secret (TS)**. Users are called **subjects**; data (table, tuple, or attribute) are **objects**.

The Bell-LaPadula model

- **Simple security property ("no read up", NRU)** — a subject can't *read* an object with a higher sensitivity label than the subject's own. E.g. a user with clearance `U` cannot view a salary value classified `C`.
- **Star property ("no write down", NWD)** — a subject can't *write* to an object with a *lower* sensitivity label than the subject's own — this prevents information flowing from a higher to a lower classification. E.g. a user with clearance `S` cannot insert a new tuple containing only `C`-classified information (that would let a lower-clearance user infer something about a higher-clearance operation via the write).

Each **tuple** gets a **Tuple Classification (TC)** — the highest classification of any of its attribute values — and, by the same rule one level up, a **table's** classification is the highest `TC` of any tuple in it. That second step is what makes the star property bite in practice: in

week12-tutorial-5-1-database-security[2], an `Operation` table whose tuples top out at `S` is an `S` object, so a `TS` subject inserting into it is writing *down* and is refused.

```
(a) EMPLOYEE (original)
Name       Salary      JobPerformance  TC
Smith (U)  40000 (C)   Fair (S)        S
Brown (C)  80000 (S)   Good (C)        S

(b) EMPLOYEE, as seen by a Confidential-clearance user
Name       Salary      JobPerformance  TC
Smith (U)  40000 (C)   NULL (C)        C
Brown (C)  NULL (C)    Good (C)        C

(c) EMPLOYEE, as seen by an Unclassified-clearance user
Name       Salary   JobPerformance  TC
Smith (U)  NULL(U)  NULL (U)        U
```

A `C`-clearance user sees Smith's `JobPerformance` as `NULL` (it's actually `S`-classified — above their clearance) and Brown's `Salary` as `NULL` (`S`-classified).

Inference attacks and polyinstantiation

Suppose a `C`-clearance user runs `UPDATE EMPLOYEE SET JobPerformance = 'Excellent' WHERE Name = 'Smith'` (working from view (b), where they only ever saw `NULL` for that field). The system **must not reject** this — rejecting it would let the user *infer* that a real, higher-classified value already exists (a **covert channel** leak). The solution is **polyinstantiation** — maintaining multiple tuples with the same key but different classifications:

```
(d) EMPLOYEE, polyinstantiated
Name       Salary      JobPerformance    TC
Smith (U)  40000 (C)   Fair (S)          S
Smith (U)  40000 (C)   Excellent (C)     C
Brown (C)  80000 (S)   Good (C)          S
```

Both the original `S`-classified fact and the new `C`-classified "fact" now coexist — a `C`-clearance user sees "Excellent"; an `S`-or-higher user still sees the original "Fair" (plus, depending on implementation, awareness that a lower-classified duplicate also exists).

[2] courses/infs1200/week12-tutorial-5-1-database-security.pdf
[3] courses/infs1200/week12-tutorial-5-1-database-security.pdf

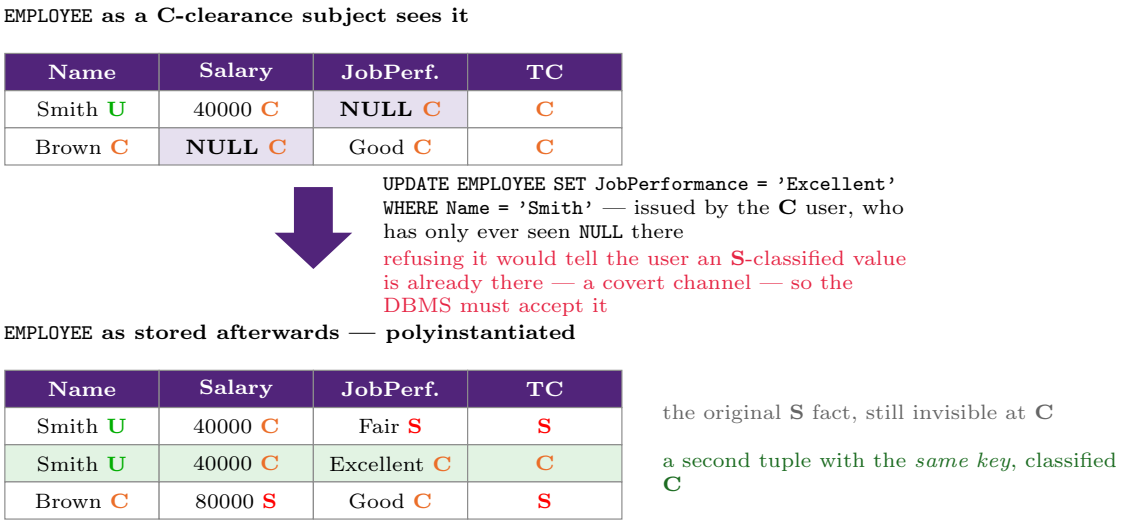

Figure 1: Why a MAC database will store two tuples with the same key. View (b) of `EMPLOYEE` sits above the stored result (d), with the offending UPDATE on the arrow between them. The point is the chain of forced moves: the C user was shown NULL, so the update is a reasonable thing for them to issue; rejecting it would tell them a higher-classified value is already there, which is the covert channel; accepting it in place would destroy an S-classified fact by writing down over it. Storing both is the only move left, and it is also why an apparently duplicate INSERT succeeds in week12-tutorial-5-1-database-security[3]'s Section B.

DAC vs. MAC

	Discretionary	Mandatory
Flexibility	High — owners choose who gets what	Low — rigid, centrally imposed
Propagation control	None — DAC doesn't restrict how information, once accessible, is further shared	Strong — enforces classification rules on every read/write
Protection	Weaker	Stronger — prevents illegal information flow

Role-Based Access Control (RBAC)

Permissions are associated with **organisational roles**, and users are assigned to the appropriate role(s) — roles can then have DAC or MAC methods applied to them as a unit.

```
CREATE ROLE manager;
DROP ROLE manager;

GRANT ROLE full-time TO emp_typ1;
GRANT ROLE intern TO emp_typ2;

GRANT privilegeName ON objectName TO {userName | PUBLIC | roleName} [WITH GRANT
 ↪  OPTION];
REVOKE privilegeName ON objectName FROM {userName | PUBLIC | roleName};
```

A typical hierarchy: **users** → **groups** → **roles** → **privileges** (e.g. `alice` is in group `admin`, which maps to `admin_role`, which grants `ALL ON SERVER server1`). RBAC's flexibility and easier administration make it a popular choice for web-based applications.

Other control measures

Inference control

Statistical databases provide aggregate statistics about a population (e.g. for government statisticians or market researchers) without exposing individual-level confidential data — only **statistical queries** using aggregates (`COUNT`, `SUM`, `MIN`, `MAX`, `AVG`, standard deviation) are permitted:

```
SELECT COUNT(*) FROM PERSON WHERE <condition>;
SELECT AVG(Income) FROM PERSON WHERE <condition>;
```

The risk: a narrow enough `WHERE` condition can isolate a group small enough (even a single individual) that an "aggregate" query effectively discloses that person's data. Mitigations:

- **k-anonymity** — enforce a minimum threshold on the number of tuples any query's result can be based on.
- **Prohibit sequences of queries** that all refer to the same (or overlapping) population of tuples, which together could be combined to isolate an individual.
- **Differential privacy** — introduce carefully calibrated noise/ inaccuracy into results.

Flow control

Regulates the distribution of information among accessible objects, verifying information doesn't flow — explicitly or implicitly — into *less* protected objects (this is exactly the Bell-LaPadula "no write down" idea, generalised). A **flow policy** specifies which channels information may move along — e.g. in a simple confidential (C) / nonconfidential (N) scheme, flow from C to N is prohibited. Example: an income tax computing service might be allowed to retain a customer's address, but not their income/deductions data.

Data encryption

Encryption converts data into ciphertext, using an encryption algorithm and a key; recovering the original data requires the corresponding decryption key.

- **DES (Data Encryption Standard)** — developed by the US government for general public use.
- **AES (Advanced Encryption Standard)** — a newer, more difficult to crack standard.

SQL injection

SQL injection is one of the most common threats to a database system: an attacker injects string input through an (often web-facing) application in a way that changes or manipulates the resulting SQL statement to their advantage.

Methods

- **SQL manipulation** — changes an existing SQL command, e.g. adding conditions to a `WHERE` clause. Classic target: the login form.
- **Code injection** — adds *entirely new* SQL statements/commands by exploiting improper handling of untrusted input.
- **Function call injection** — inserts a database or OS function call into a vulnerable SQL statement, to manipulate data or make a privileged system call.

SQL manipulation example

A naive login check builds its query by directly concatenating user input:

```
SELECT * FROM Users WHERE Username = '<input>' AND Password = '<input>';
```

Normal login (`Jack` / `Pass123`) produces `WHERE Username = 'Jack' AND Password = 'Pass123'` — matches the stored row, access granted. But an attacker who knows the username `Jack` can enter the password field as:

```
' OR 'x'='x
```

giving the executed query:

```
SELECT * FROM Users WHERE username = 'Jack' AND (password = '' OR 'x'='x');
```

`'x'='x'` is always true, so the `AND` collapses to just `username = 'Jack'` — the row is returned regardless of the actual password, and access is granted.

Code injection example

Going further, appending a second statement via a semicolon:

```
wrongpass' OR 'x'='x'; drop table Users;
```

If the application naively executes whatever SQL string results (and the driver/DBMS allows statement-stacking), this doesn't just bypass login — it can **destroy the entire `Users` table**.

Risks associated with SQL injection

- **Database fingerprinting** — determining the type of database in use (to target version-specific exploits).
- **Denial of service** — denying service to valid users.
- **Bypassing authentication** — gaining access without valid credentials (as shown above).
- **Identifying injectable parameters** — learning about the backend structure through error messages/behaviour.
- **Executing remote commands** — running harmful commands remotely (e.g. via function call injection).
- **Performing privilege escalation** — upgrading the attacker's own access level.

Protection techniques

- **Bind variables (parameterized statements)** — pass user input as *bound parameters*, never concatenated directly into the SQL string. This is the primary defence: the database driver treats parameter values purely as data, never as executable SQL syntax — and as a bonus, it also improves performance (query plans can be cached and reused).
- **Filtering input (input validation)** — strip/escape characters (like unescaped quotes) that could otherwise be used to break out of a string literal and inject manipulation.
- **Function security** — restrict which standard and custom database functions are callable, to limit the blast radius of function call injection.

Summary

You should now be able to explain the threats to database security and the CIA principle; describe DAC, MAC, and RBAC and apply them (given grant/revoke commands, or a Bell-LaPadula classification scenario) to determine resulting access; and explain SQL injection, its risks, and how to defend against it. This completes the course content. See week12-tutorial-5-1-database-security[4] for practice.

Reading: the matching textbook material is the database-security chapter of Elmasri & Navathe — Chapter 30 in the 7th edition.

[4] courses/infs1200/week12-tutorial-5-1-database-security.pdf