Database Security

Security 1 (CM0475, CM0493) 2020-21
Università Ca' Foscari Venezia

Riccardo Focardi
www.unive.it/data/persone/5590470
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Motivations

What makes database security relevant

Databases tend to **concentrate sensitive information** in a single point:

- Financial data
- Personal data of customers
- Proprietary product information (IP)
- Medical records
- ...

Motivations

What makes database security difficult

- DataBase Management Systems (DBMS) are very **complex**
- Databases offer a complex access language: ***Structured Query Language*** (***SQL***)
- Real systems often **integrate** different DBMS technologies running on various operating systems

Motivations

What makes database security different

Databases need **dedicated** access control systems and security mechanisms

- regulate access to specific **records** and **fields** in the database
- deal with the peculiarities of ***Structured Query Language*** (***SQL***)

Relational databases

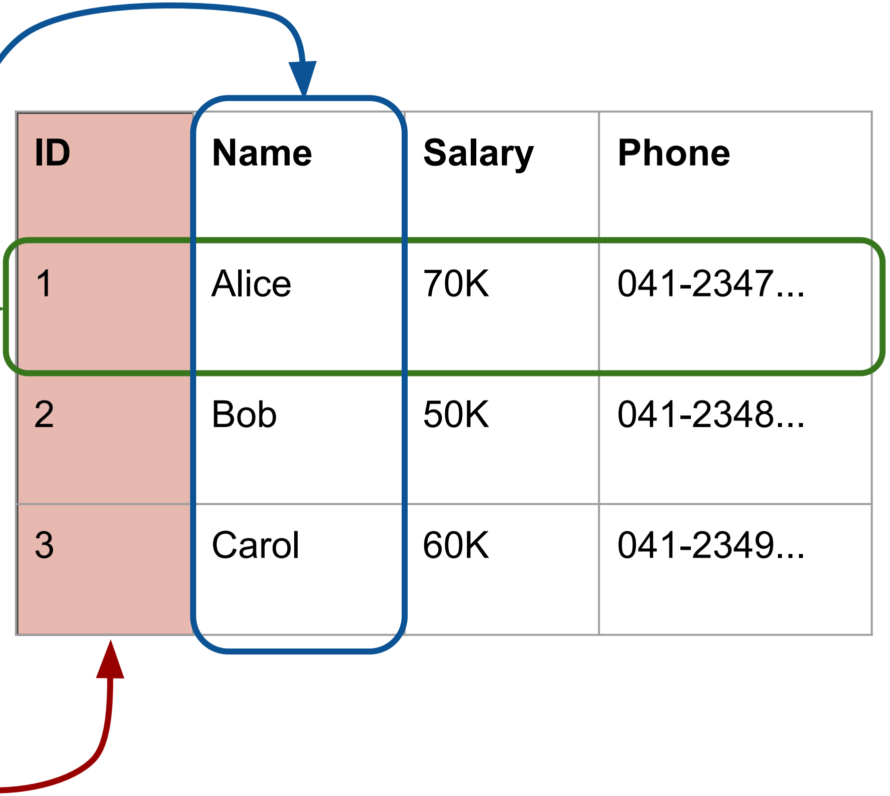
Table: a *relation* in the form of a N x M matrix

Field: a *column* of the table

Record: a *row* of the table

Primary key: one or more fields (columns) that uniquely identify a record (row)

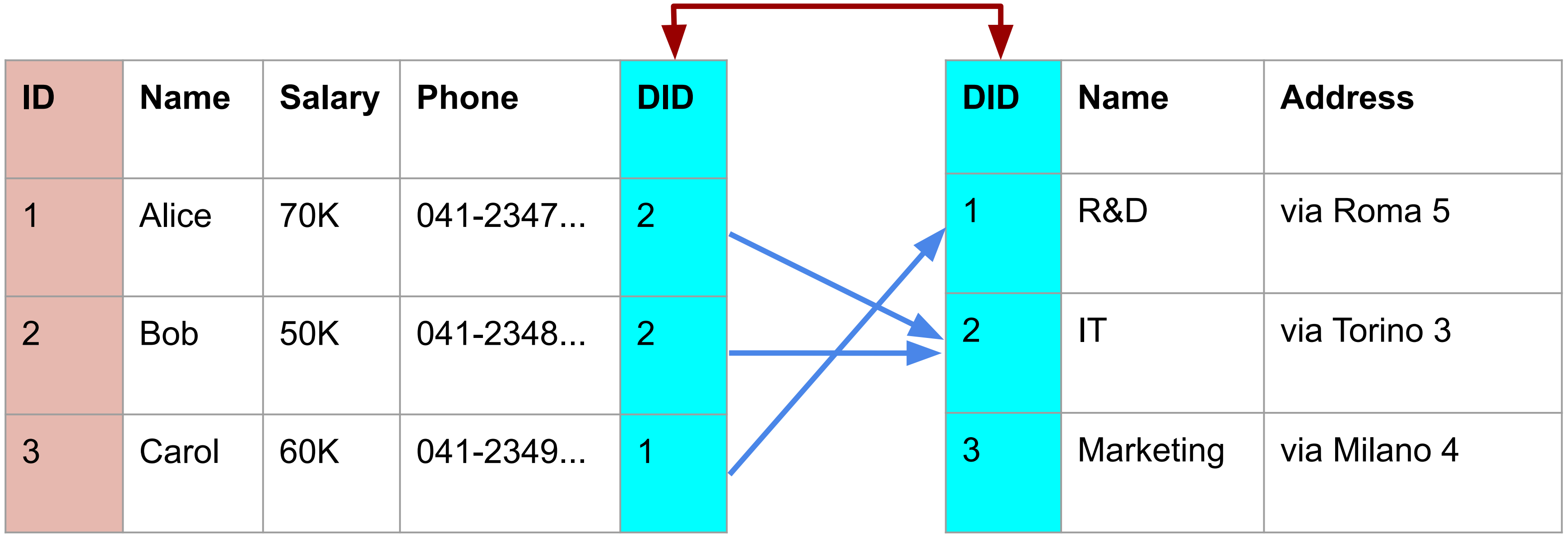
- Typically a unique ID



ID	Name	Salary	Phone
1	Alice	70K	041-2347...
2	Bob	50K	041-2348...
3	Carol	60K	041-2349...

Relationships

foreign key: a primary key of one table appearing as field of another table



ID	Name	Salary	Phone	DID
1	Alice	70K	041-2347...	2
2	Bob	50K	041-2348...	2
3	Carol	60K	041-2349...	1

DID	Name	Address
1	R&D	via Roma 5
2	IT	via Torino 3
3	Marketing	via Milano 4

Views

View: a virtual table with **selected rows and columns** from **one or more tables**

Can be used for security to give a **partial view** of data

Example: Employees with department name, address, phone number (salary is hidden)

Name	**DName**	**Address**	**Phone**
Alice	IT	via Torino 3	041-2347...
Bob	IT	via Torino 3	041-2348...
Carol	R&D	via Roma 5	041-2349...

Structured Query Language (SQL)

SQL: a standardized language that can be used to

- create tables
- insert and delete data in tables
- create views
- retrieve data with query statements

```sql
CREATE TABLE Employee (
   ID INTEGER PRIMARY KEY,
   Name CHAR (30),
   Salary INTEGER,
   Phone CHAR (10),
   DID INTEGER,
   FOREIGN KEY (DID)
      REFERENCES Department (DID)
)

CREATE TABLE Department (
   DID INTEGER PRIMARY KEY,
   Name CHAR (30),
   Address CHAR (60)
)
```

SELECT and VIEW

SELECT statements extract data satisfying constraints

```
SELECT Name, Phone
    FROM Employee
    WHERE DID = 2
```

Name	Phone
Alice	041-2347...
Bob	041-2348...

VIEW is an abstract table built through a SELECT statement

```
CREATE VIEW EmplDep
        (Name, Dname, Phone)
  AS SELECT E.Name, D.Name, E.Phone
    FROM Department D Employee E
    WHERE E.DID = D.DID
```

Name	DName	Phone
Alice	IT	041-2347...
Bob	IT	041-2348...
Carol	R&D	041-2349...

SQL injection

(SQLi)

SQLi, along with injection attacks, is considered the **top web application security threat** [OWASP Top 10]

Injection attack: the attacker triggers unexpected behaviour by supplying untrusted, **malicious input** to an application

SQLi scenario

Web applications

- have **dynamic content** that depends on data stored in databases
- manage data through **queries**

⇒ When queries depend on **untrusted user input** an attacker might **inject malicious SQL code** that will be sent to the database

Typical attack:

1. Attacker sends **malicious input**
2. The web application server executes a query that contains the input (**injection**)
3. The result of the query is **included** in a dynamic web application page
4. Attacker gets **sensitive data** directly from the web page

SQLi example

Attacker inject input that

1. **terminates** a string with a quote
2. adds **malicious** code
3. **comments out** the rest of the query (including the original closed quote)

Example:

```
Query = "SELECT * FROM Users WHERE
   Name = '" + Username + "'"
```

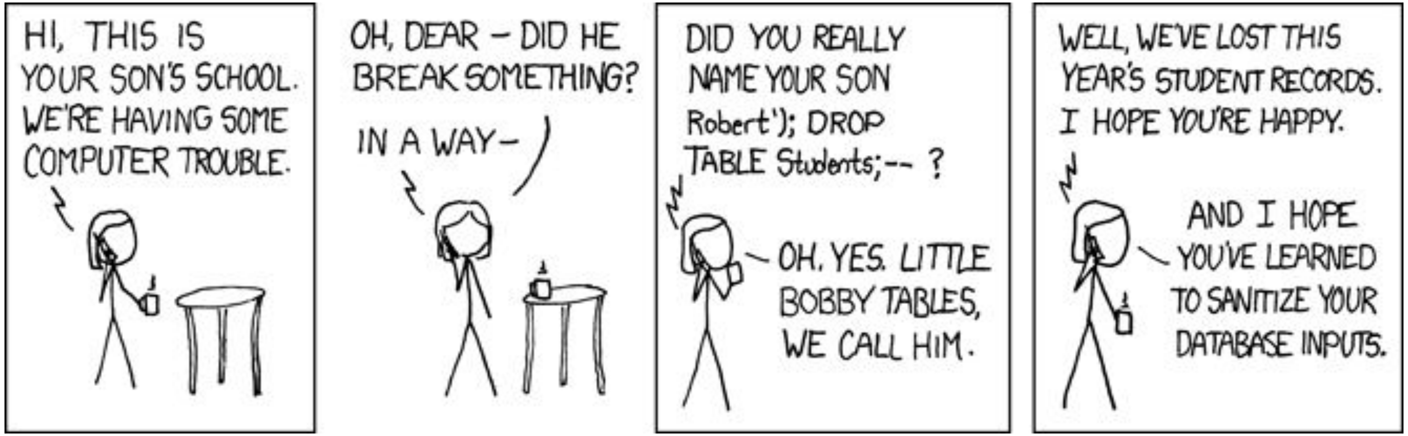
where `Username` is the (untrusted) input taken from a web form

```
Username = "'; DROP TABLE Users -- "
```

will give:

```
SELECT * FROM Users WHERE
   Name = ''; DROP TABLE Users-- '
```

Note: In **mysql** "`--`" should have a space before the comment, as in "`-- `"


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

“Exploits of a Mom” https://xkcd.com/327/

Origins of injection

User input: input from **forms** is used to compose SQL queries

Server variables: **headers** that are logged and might be modified by the attacker. For example, headers logged for usage statistics

Second-order injections: the attacker injects data **in the database** that is, in turn, used to compose another query

Cookies: browser cookies are used to implement stateful sessions, but can be manipulated by the attacker. This can trigger injections when **cookie value** is used to compose queries

Physical user input: input that comes from physical **devices** or **media**. Examples are barcodes, RFID tags, scanned paper documents, ...

SQLi

Attack types

Inband: uses the **same communication channel** for SQLi and retrieving results

Inferential: no direct leakage; the attacker reconstructs the information by **observing the resulting behavior**

Inband attacks (1)

Tautology: This form of attack injects code in conditional statements so they **always evaluate to true**

Example: authentication check

```
Query = "SELECT * FROM Users WHERE
   Name = '" + Username + "' AND
   Pwd =  '" + Password + "'"
```

Authentication fails if the query returns an empty result

The attacker injects

```
Username = "admin"
Password = "' OR 1=1 -- "
```

which makes the **WHERE** condition always true

```
SELECT * FROM Users WHERE
   Name = 'admin' AND
   Pwd =  '' OR 1=1 -- '
```

⇒ Attacker logs in as **admin**!

Inband attacks (2)

End-of-line comment: legitimate code that follows is **nullified** through usage of end of line comments

Example: same as before ...

```
Query = "SELECT * FROM Users WHERE
   Name = '" + Username + "' AND
   Pwd =  '" + Password + "'"
```

Authentication fails if the query returns an empty result

The attacker injects

```
Username = "admin' -- "
Password = ""
```

which **nullifies** the `AND` condition

```
SELECT * FROM Users WHERE
   Name = 'admin' -- ' AND Pwd =  ''
```

⇒ Attacker logs in as **`admin`**!

Inband attacks (3)

Piggybacked queries: The attacker adds **additional queries** beyond the intended query, *piggybacking* the attack on top of a legitimate request

NOTE: This technique relies on server configurations that **allow for different** queries within a single string of code

As seen before, the attacker injects

```
Username = "'; DROP TABLE Users -- "
Password = ""
```

which *piggybacks* a `DROP` request

```
SELECT * FROM Users WHERE
   Name = ''; DROP TABLE Users -- '
   AND Pwd =  ''
```

⇒ Attacker drops a table!

Inferential attacks

Incorrect queries: the default **error page** returned by application servers is often overly descriptive, revealing

- the **query** (or a significant part of the query)
- name of **tables** and **columns**
- possible input **filtering**

⇒ Typically **the first step of attacks**

Blind SQL injection: attacker infers the data present in a database even when the application **does not display** errors or data

The attacker "asks the server" **true/false questions** and observes the behaviour. Example

- User logs in
- Authentication is refused

SQLi

Countermeasures

Defensive coding: secure coding principles that **prevent SQLi**

Detection/prevention: **detect** and **block** attacks at runtime, e.g., *Web Application Firewalls* (*WAF*)

Testing: tools that **search** for SQLi vulnerabilities (pentest tools)

Defensive coding

Whitelisting input: check that input belongs to a whitelist of **trusted values**
Example: a column name for sorting

Strict typing: check input **type**
Example: integer values

Prepared statements: query is **parametrized** and pre-parsed; parameters never interpreted as code

Typed APIs: generic APIs for DBMS access with (typed) **parameterized queries**. Example: PHP PDO

Trusted input: crypto mechanisms to ensure **input authenticity**. Example: ***HMAC*** for cookies, RFID, barcodes

Sanitization: use **standard** functions to **sanitize** input. Last resort, when no other defence is possible

Prepared statements example

```
mysql> PREPARE stmt1 FROM 'SELECT * FROM people WHERE lastname=?';
Statement prepared

mysql> set @n = 'focardi';

mysql> EXECUTE stmt1 USING @n;
+----+-----------+----------+----------+-----------------------+------------+----
| id | name      | lastname | username | mail                  | password   | url
+----+-----------+----------+----------+-----------------------+------------+----
|  2 | Riccardo  | Focardi  | r1x      | focardi@dsi.unive.it  | ********** | htt
+----+-----------+----------+----------+-----------------------+------------+----

mysql> set @n = "'' OR 1 # ";

mysql> EXECUTE stmt1 USING @n;
Empty set (0.00 sec)
```

Statement is parsed and prepared

Trying the injection

Injection fails: SQL has been parsed already and data are only interpreted as data

Database Access Control

Control access to specific **portions** of the database

Access rights might be determined by the **values** (e.g. through views)

DAC and **RBAC**

Managing privileges

Grant: used to grant access on specific tables to users/roles

Example:

```
GRANT SELECT ON * TO alice
```

⇒ Grants `SELECT` (**read**) access on the whole database to user `alice`

Revoke: used to revoke access rights previously granted

Example:

```
REVOKE SELECT ON * FROM alice
```

⇒ Revokes the previously granted permission

Delegation and cascading

Privileges granted with “grant” option can be, in turn, granted to more users

Example:

```
GRANT SELECT ON * TO alice
WITH GRANT OPTION
```

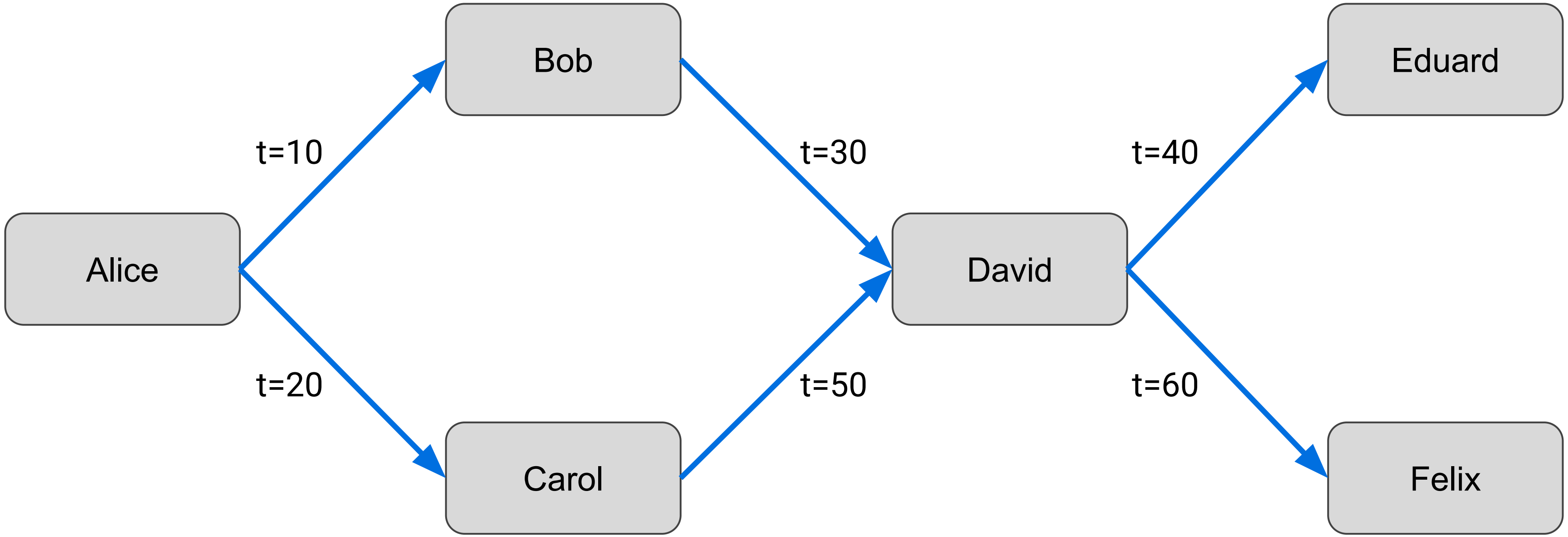
delegates `alice` to grant the same permission to `bob`, `carol`, ...

Some DBMS implements revoke **cascading**

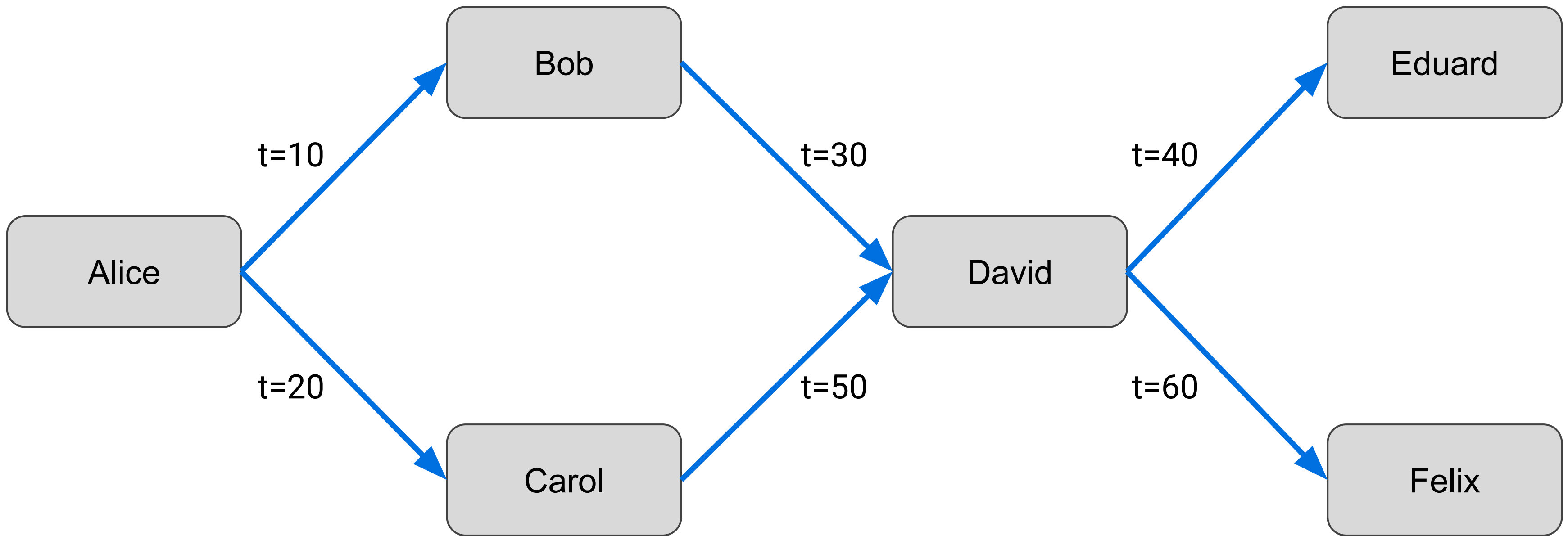
```
REVOKE SELECT ON * FROM alice
CASCADE
```

revokes the permission from `alice` and from **all the users** who got the permission through an `alice`’s grant

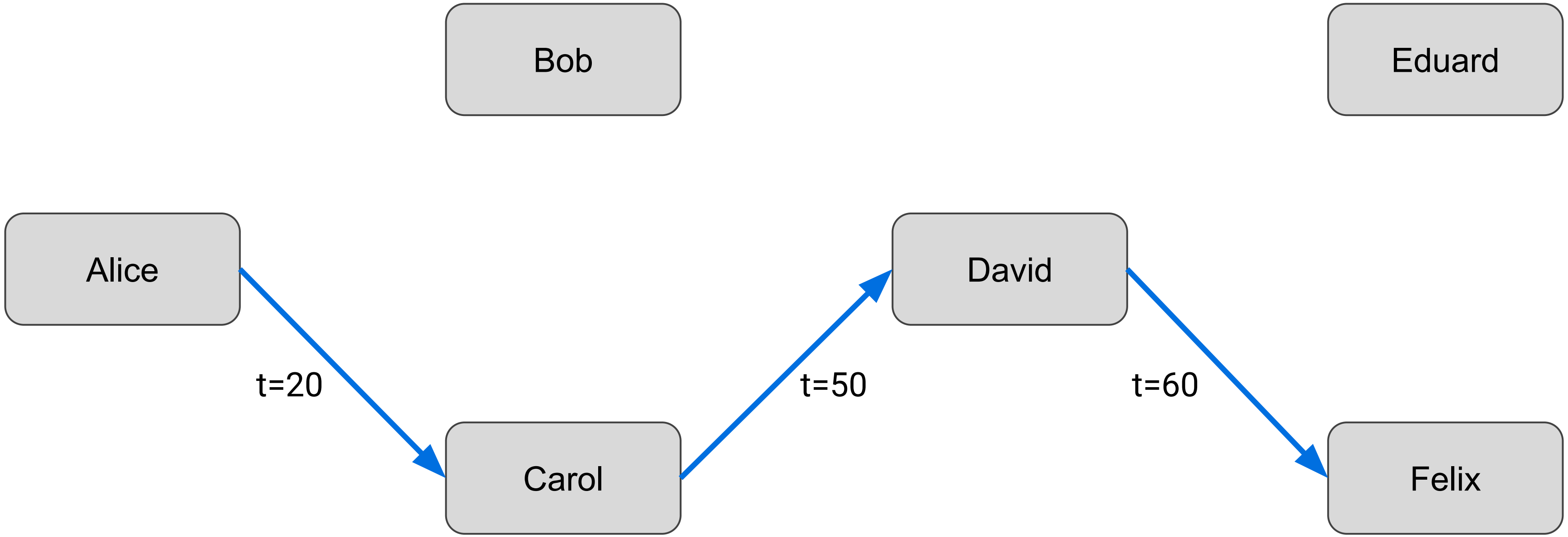
Example: cascading



Example: cascading



Example: Alice revokes grant to Bob



Roles: example

```
CREATE ROLE 'app_developer', 'app_read', 'app_write';

GRANT ALL                      ON * TO 'app_developer';
GRANT SELECT                   ON * TO 'app_read';
GRANT INSERT, UPDATE, DELETE ON * TO 'app_write';

GRANT 'app_developer' TO 'dev1';
GRANT 'app_read'      TO 'read_user1', 'read_user2';
GRANT 'app_read', 'app_write' TO 'rw_user1';
```

- `rw_user1` can **SELECT**, **INSERT**, **UPDATE**, **DELETE**
- `read_user1` and `read_user2` can only **SELECT**