

# IT Security

## Database Authorization

Nikolaus Augsten

nikolaus.augsten@sbg.ac.at

Dept. of Computer Sciences  
University of Salzburg

Winter Semester 2015/16

## Table of Contents

- 1 Introduction
- 2 Access Control Models
- 3 Authorization in SQL
- 4 Application Security

## Table of Contents

- 1 Introduction
- 2 Access Control Models
- 3 Authorization in SQL
- 4 Application Security

All infos about the database part in this lecture

<http://dbresearch.uni-salzburg.at/teaching/2015ws/its/>



## Acknowledgments

The sections “Authorization in SQL” and “Application Security” are adapted with kind permission from Sven Helmer’s slides on these topics:

<http://www.inf.unibz.it/dis/teaching/DBS/>

## Table of Contents

- 1 Introduction
- 2 Access Control Models
- 3 Authorization in SQL
- 4 Application Security

## Access Control Models

- Discretionary Access Control (DAC)
  - File permissions in Unix (read/write/execute for user, group, and others)
  - ACL: Access Control List (supported by Windows since NT and many Unix file systems)
  - RBAC: Role Based Access Control (supported by many database systems)
- Mandatory Access Control (MAC)
  - allows policies to be enforced
  - safer than DAC for sensitive information
  - governmental and military use

## Table of Contents

- 1 Introduction
- 2 Access Control Models
- 3 Authorization in SQL
- 4 Application Security

## Authorization

- A user may be assigned authorizations on parts of a database
- Authorizations cover
  - reading data
  - inserting new data
  - updating data
  - deleting data
- Each type is called a *privilege*
- A user may have all, none, or a combination of privileges (for parts of a DB)

## Granting Privileges

- Privileges can be granted to a user ...
- ... and later on be revoked again
- One user, the *database administrator*, has all the privileges
- Granting and revoking privileges is done via SQL commands
  - This is part of the Data Definition Language (DDL)

**NOTICE**



## SQL Syntax

- The general statement for granting privileges is:
 

```
grant privilege list
on relation or view name
to user or role list;
```
- A *privilege list* is made up of a combination of **select**, **insert**, **update**, and **delete**
  - ... or **all privileges** for all of them
- This is followed by a relation or view name
- and a user name (we'll come to roles in just a moment)

## Examples

- **grant** select  
  **on** student  
  **to** peter, paul, mary;
- The users peter, paul, and mary may run select queries on the relation student
- When granting update and insert privileges, attributes can be specified:
 

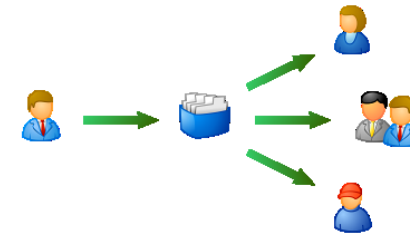
```
grant update(office_no)
on professor
to peter;
```
- This allows the user peter to update the attribute `office_no` in the relation professor

## Revoking Privileges



- Privileges can also be withdrawn via a `revoke` statement
- The general syntax is:  
**revoke** *privilege list*  
**on** *relation or view name*  
**from** *user or role list*;
- Works like a `grant` statement in reverse

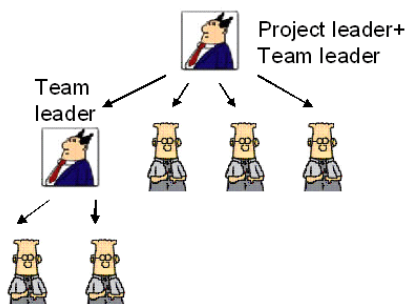
## Multiple Users



- Large database system may have hundreds or even thousands of users
- Granting and revoking privileges individually on all relations may be very tedious
- The user name `public` grants a privilege to every user of the system
- A more fine-grained approach uses *roles*

## Roles

- Often groups of people do similar work and need the same privileges
- In a database it is possible to
  - define a role
  - give privileges to this role
  - and add users to this role



## SQL Syntax

- Here are some examples on how this looks in SQL:

```
create role instructor;
```

```
grant select  
on course  
to instructor;
```

```
grant instructor to john;
```

```
create role professor;
```

```
grant instructor to professor;
```

```
grant professor to sven;
```

## Authorization and Views

- Privileges in combination with views can be used to make parts of a relation visible
- For example, an administrator may only see records of computer science assistants
  - Create the following view:
 

```
create view csasst as
select *
from assistant
where area = 'computer science';
```
  - Then grant select privilege on csasst and revoke all privileges on base table assistant

## Transfer of Privileges [1]

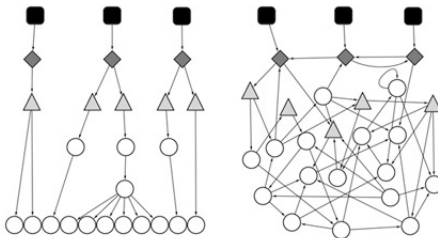
- A user who has been granted a privilege may be allowed to pass it on
  - The default does not allow this
- If we want to allow someone to grant a privilege to others, we use the **with grant option**

```
grant select
on student
to peter with grant option;
```



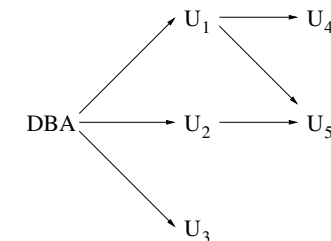
## Transfer of Privileges [2]

- Usually the creator of a database object holds all privileges
  - This includes the privilege to grant privileges
- What happens if there is a whole chain of granted privileges and we start revoking some?



## Authorization Graph

- We can use an *authorization graph* to check:



- A user has a privilege, iff there is a path from the root (DBA) to the user node
- Revoking a privilege from a user
  - removes that user
  - and everyone on outgoing edges of that user not connected to the root otherwise

## Cascading Revokes [1]

- Revoking a privilege from  $U_1$  from the previous graph
  - also removes  $U_4$ 's privilege
  - but not  $U_5$ 's, as he/she is still connected via  $U_2$
- Recursively revoking privileges is called a *cascading revoke*
- Can be prevented by the `restrict` clause
  - Will return an error if there is a cascading revoke

## Cascading Revokes [2]



- Sometimes privileges should be granted by a role, not an individual
- For example, the role of *dean* can grant privileges associated with the role of *professor* or *instructor*
  - If the current dean steps down and the user account is removed, granted privileges should stay
  - Can be done by adding the clause `granted by current_role`

## Limits of Authorization in SQL [1]

- While SQL supports a fairly flexible system, it has limits
- Many applications require a very fine-grained authorization
- For example, we want students to see only their own grades
- That means, we need authorization on the tuple level
  - Databases only support relation, view, or attribute level



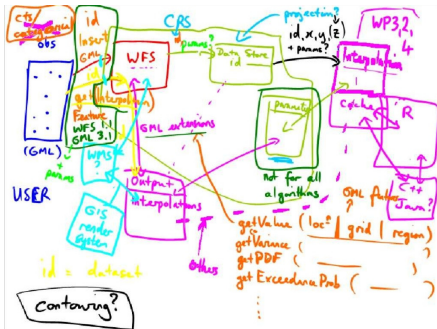
## Limits of Authorization in SQL [2]

- Often, there is a lack of end-user information
- For example, in web applications end users usually do not have individual user IDs in the database
- Makes it difficult to apply the SQL authorization scheme



## Limits of Authorization in SQL [3]

- As a consequence a lot of the authorization moves into the application code
- The point of a DBS was to provide infrastructure and have clear responsibilities



## Table of Contents

- 1 Introduction
- 2 Access Control Models
- 3 Authorization in SQL
- 4 Application Security

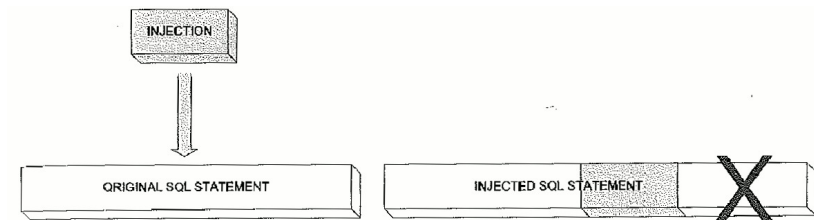
## Application Security

- Even if the database is pretty secure, a badly written application can compromise the whole system
- Many database applications have a web (or mobile) interface that can be exploited
- In particular, we are looking at
  - SQL injection
  - Cross-site scripting and request forgery



## SQL Injection [1]

- In SQL injection attacks, the database runs an SQL query created by an attacker
- This is usually done by manipulating a valid SQL statement:



## SQL Injection [2]

- Applications that build SQL queries on the fly are especially vulnerable to this
- For example, assume a Java application gets a string name and constructs the query

```
"select * from student where name = '"+name+"';"
```

- Instead of a name, a user might enter some SQL:

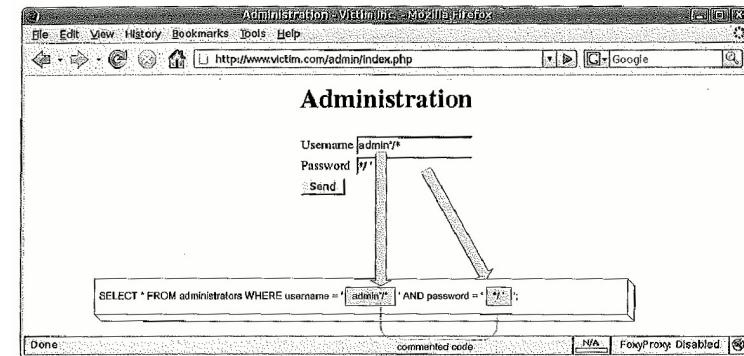
```
X' or 'Y' = 'Y
```

turning the SQL statement into

```
select * from student
where name = 'X' or 'Y' = 'Y';
```

## SQL Injection [3]

- Depending on the application, this can have serious consequences:



- Here comments are used to cut part of the SQL query

## SQL Injection [4]

- This is not just limited to select statements
- Depending on the configuration of the server, multiple statements may be executed in one go



Source: <http://xkcd.com/327/>

## Remedies

- So, how should you build your database application?
- Any query that relies on user input should use prepared statements
- In prepared statements, some values are replaced by "?"
- For example, the following will insert a tuple

```
PreparedStatement pSt = con.prepareStatement(
    "insert into student values (?, ?, ?)");
pSt.setInt(1, 102093);
pSt.setString(2, "James Smith");
pSt.setDate(3, "1991-10-05");
pSt.executeUpdate();
```



## Prepared Statement

- Not only will this run faster (if SQL statement is used multiple times)
- It will also escape special characters
- For example, the string

`X' or 'Y' = 'Y`

would become

`X\' or \'Y\' = \'Y`

rendering the attempted attack harmless

## Other Forms of Attack

- Not every attack can be prevented with prepared statements
- For example, the following lets a user sort a result:  
`"select * from student order by "+orderAtt+";"`
- Application has to make sure that the variable `orderAtt` can only contain valid attribute names
- In general, any input coming from a user has to be sanitized!

## Cross-Site Scripting (XSS) [1]

- Many web sites rely on the execution of code embedded in HTML on the client side
  - Client-side scripting languages such as JavaScript are a popular option
- If an attacker is able to smuggle code onto a web site, it may be executed on a client
- For a database-related example, assume the following:
  - Users enter data into a database via a web site
  - Later on, other users view this information
  - Malicious users can enter JavaScript instead of data

## Cross-Site Scripting (XSS) [2]

- The effects of executing malicious code include
  - changing or deleting files on the local system
  - monitoring key strokes
  - sending out confidential information (e.g. cookies)
  - interacting with other web sites of a user

## Cross-Site Request Forgery (XSRF)



- XSRF attempts to hijack a session running in another tab or window of the browser
- Can fool a server, as request is coming from a valid client
- Can even be done without scripting, e.g.  

```

```

## Protection from XSS/XSRF

- We provide some general remarks (there are more complex attacks)
- Preventing your site from becoming an attack launch pad:
  - Sanitize all user input
  - There are functions to strip out HTML, scripts, or other code
- Preventing your site from becoming a target:
  - Check referer in the HTTP header
  - Tie session not only to cookies, but also to IP address
  - Never use GET to update any data or to send sensitive data

## Password Leakage



- Storing passwords in clear text in application code or a database is not a good idea
- If you have to store a password, it needs to be encrypted
- Many databases can be configured to use authentication scheme of operating system