Lecture 7 — Security

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 11, 2023

Security, Report to the Bridge

In a lot of books, and even in my original plan for how the lectures would be carried out, security was left to the end.

Security is something you want to bake into your product and something you want to have in your mind when you are designing it.

It doesn’t work to try to bolt it on afterwards.

Terminology

An attacker is a malicious user who is trying to damage or exploit the system.

A countermeasure is some action or process we can take or implement to defend against attacks.

Security Policies

In this lecture we will not go into the legal or ethical issues related to security.

We also will not discuss security policies that are designed by organizations to tell you what keeps the system secure.

We will focus on how access is controlled in the database, and specifically about the SQL Injection attack.

Sensitive Data

The truth is that databases contain a lot of data and some of it might be sensitive.

It might be that it is personal information, such as medical history of a patient or an employee's home address.

It could also be that it comes from a sensitive source.

The sensitive data might not be the entire tuple; it might just be a particular attribute such as Social Security Number or salary.

Sensitive Data

Regardless of why some data is sensitive, database administrators must see to it that the security policies are enforced.

This means specifically that sensitive data is protected, but also that data is not corrupted and that access to any data is limited to those who should have access.

Security policies do have some tradeoff with usability.

You most certainly do NOT want to find your company's name on TV having to report a data breach in which user personal data was stolen.

Access Control

We will talk about access control.

That is, what users have access to what in the database.

If this is configured correctly, then a lot of the security (although not all) is handled by the database server.

But like seatbelts, access control doesn't work if you don't use it.

The DBA

It is the job of the database administrator, mostly, to control accounts and privileges.

This is analogous to the administrator or root or superuser account in operating systems, in that they have permissions to do everything and anything...

But it is not good practice to log in using that account...

It is better to create limited accounts and assign only the exact privileges needed to those accounts...

Username and Password

Database systems have accounts and user accounts have passwords.

Users are expected to log in with their names and passwords when database access is needed.

The user name and password will be checked by the database server and if they match, the user is granted access. Otherwise, access denied.

Storing Internally

The database itself will likely use some tables internally.

When accounts and permissions are created they are recorded in the database's internal tables.

After all, the data could be organized using the toolset we already have... relational database tables.

Audit Trail

When a database user has successfully logged in, they have a login session.

The session is associated with all of the interactions that user does while logged in.

This can be used to track who has logged in and who made what changes so that administrators can see the history in case something has gone wrong.

If any malfeasance is suspected, database administrators can audit the logs (look through them) and determine what happened.

Granting Permissions

Privileges can be given out at the account level: and includes the create schema, create table, create view, alter, drop, modify, select...

Users are created with syntax like: `CREATE USER 'exampleuser'@'server' IDENTIFIED BY 'weakpassword';`

To delete a user, `DROP USER exampleuser;`.

To change a user's password, `SET PASSWORD FOR 'exampleuser'@'server' = PASSWORD ('CorrectHorseBatteryStaple');`

Granting Permissions

The next level is at relation level.

Relations typically have an owner, usually the account used when the relation was created.

It is likely that when a database schema is being created for the first time, an administrator account (eg root) is doing the creating.

And the owner can allow other users to access it by granting them access.

Granting Permissions

The keyword for this is `GRANT` and we'll soon see how it all works.

The following privileges may be granted:

- **Select**
- **Modification**
- **Reference Privilege**

Obviously, if privileges can be granted, they should also be able to be revoked. For that purpose, of course, there is the `REVOKE` command.

Using Grant

Our basic syntax is as follows: **GRANT** <privilege list> ON <relation> TO <user list>;

So if we want to allow selection for user Alice on the table for books, it is `GRANT SELECT ON books TO alice;`.

We can be as specific as we want to be: `GRANT UPDATE books( price ) TO bob, charlie, donna;`

If we didn't specify any particular attributes, it would be possible to update all attributes of the relation.

We can also grant `ALL` to give all privileges.

The Public

It is possible to grant permissions to a special username, “public”, which assigns the permission to all current and future users of the system.

I really recommend against this sort of broad authorization.

It goes against the idea of the principle of least privilege: user accounts should have only the privileges they need and nothing more.

Revocation

The revocation basic syntax is as follows: **REVOKE** <privilege list> ON <relation> FROM <user list>.

So if we no longer want to allow selection for user Alice on the table for books, it is `REVOKE SELECT ON books FROM alice;`.

Why Reference?

Are you wondering why we might ever want to grant the reference privilege?

Foreign key constraints restrict deletion and update operations on the referenced relation.

If someone creates a table *B* referencing another *A*, then an attempt to delete an element from *A* may fail due to the foreign key constraint added by *B*.

Thus, it is sensible to have the references privilege, since the ability to add a foreign key restricts future activity by other users.

Logging in as Root

It is easy to just ignore all this and allow all your users to have all permissions, but this is dangerous.

In particular, you might not want to allow developers this access either.

Manual changes to the database should probably be done via some sort of double-check system.

Roles

Giving out permissions individually to users might be rather tedious.

Every time a new user is added it might be necessary to run hundreds of grant statements to that new user for each table.

To manage the complexity, we might want to use roles.

RBAC

In Role-Based Access Control (RBAC), a set of roles is created, users are assigned roles, and access is granted or denied based on the role(s) a user has or lacks.

Thus, assigning rights to users is done by assigning roles to users.

A user can have more than one role, but must have at least one.

An advantage this has over direct management of permission granting is that assignment is simpler.

Roles in SQL

Roles are created very simply in SQL: `CREATE ROLE accounting;` would create the role of accounting but it would not have any permissions.

Then, we can grant permissions to this role as if it was a user: `GRANT SELECT ON payroll TO accounting;` would do the job.

But then you need to give some users the role: `GRANT accounting TO leslie;`.

Now Leslie will have all the privileges that are granted to the accounting role, including ones added to the role after it has been granted to Leslie.

Role Relations

There can also be relations between roles: doctors might be able to do all the things a nurse can do.

Rather than have extra permissions assigned everywhere, the system can be set up so the doctor role *subsumes* the nurse role.

The doctor role has all the rights of the nurse role, and may have others.

To do that, grant one role to the other: `GRANT nurse TO doctor;`

Delegation

It is also possible to delegate – pass the authorization on to others – if the permission for that is also granted.

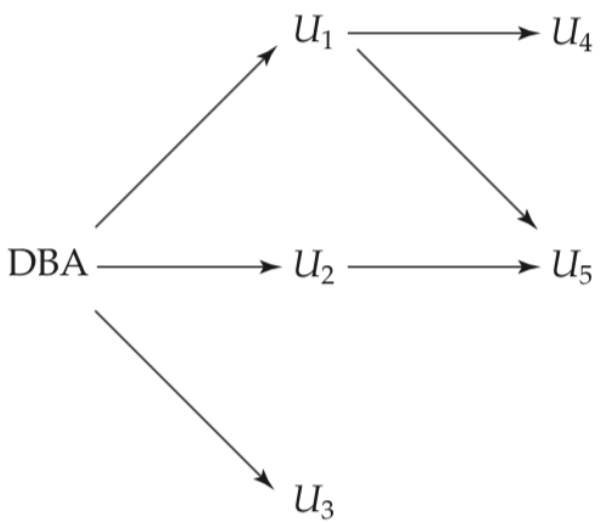
In the examples we have seen so far, the privileges cannot be granted by the recipient to any other users.

If, however, we added the `WITH GRANT OPTION` at the end of the `GRANT` command, it means that the recipient may then grant that permission.

Alice would grant to Joe by issuing a grant command herself.

Authorization Graph

The database server will want to keep track of what permissions have been granted to which users and by whom.



Authorization Graph

The graph can be checked for the existence of privileges.

A user *U* has a privilege *p* if and only if there exists some path from the root (DBA account or root user) to *U* (however many steps that takes).

This is probably inefficient, as searching the tree on every transaction to see if the user doing the command has permission to do it is likely to take too long.

Authorization Graph

What the graph is actually for is revocation.

If in the diagram, U_1 has the permission revoked by the DBA, the permission that U_1 has granted to U_4 and U_5 will be revoked as well.

This, in a practical sense, has no impact on U_5 because the permission has also been granted to U_5 by U_2, but U_4 can no longer perform the operation.

Circular Argument is Circular

You might think that you can trick the database!

If U_8 grants the permission to U_9 and then U_9 grants it to U_8, if the database administrator revokes the permission grant to U_8, what happens?

If we just used reference-counting, we would see that these users both have the permissions.

But in constructing the graph we would see the problem immediately.

Cascading Revocation

There is the possibility to modify the revoke command to prevent cascading revocation; in that case just put `RESTRICT` at the end of the statement.

If that is the case, then if the permission has been granted to anyone else (i.e., it would cascade), the revocation will not happen and the system will return an error.

The opposite of that is the `CASCADE` keyword, but it's unnecessary to use it because it is the default behaviour.

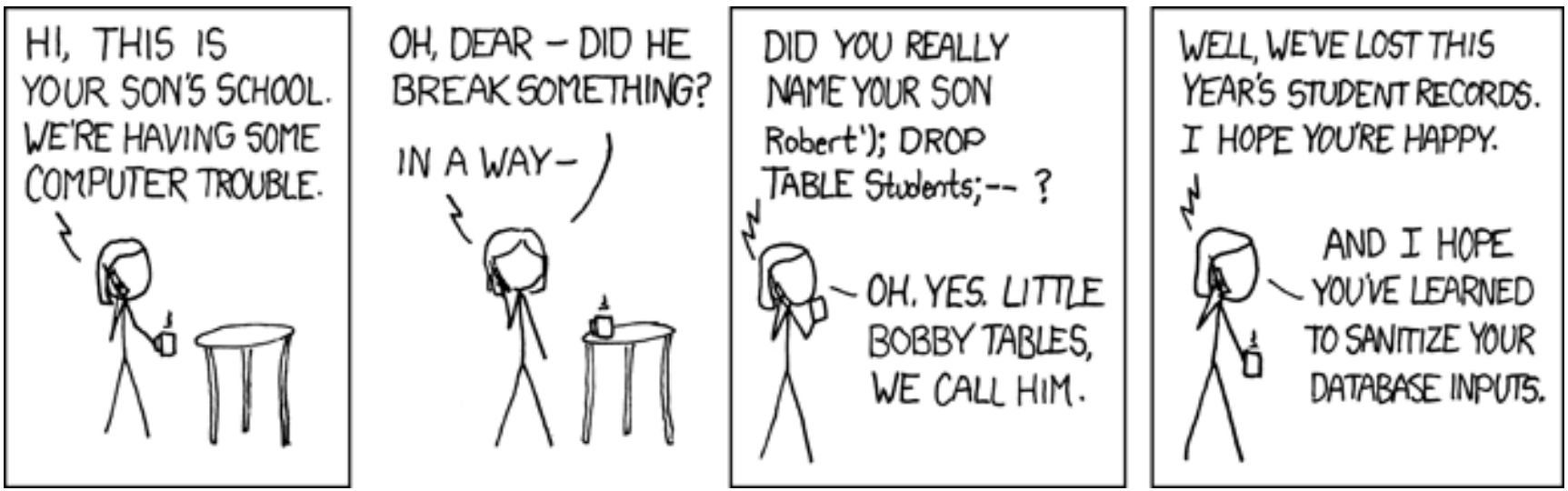
Granting from a Role

Sometimes we want to grant permission not from the current use but from a particular role.

If that is the case, then add `GRANTED BY CURRENT_ROLE` to the grant statement.

Allowing users to grant roles is, in my opinion, fairly dangerous...

SQL Injection



There are a few categories of things that can be accomplished by SQL injection:

- **SQL Manipulation**
- **Code Injection**
- **Function Call Injection**

SQL Injection

And what can you do with this sort of access?

- Execute arbitrary commands on the database (as we have seen).
- Determine what type of database is in the backend so we can see what is vulnerable.
- Denial of service attacks.
- Skip authentication (as above).
- Learn more about the internal structure of the application (mostly because error pages are very verbose).
- Privilege escalation – gaining access to things that you should not be able to access.

Avoiding SQL Injection

In general the solutions are pretty well understood: user input should be filtered (sanitized) and not allowed to be put directly into the queries.

This alone might not be enough, and it is better to bind the user input to parameters rather than dynamically building query strings.

In this course we don’t talk about how to write an application program that interacts with the database (we just tell the database what we want).

But do keep this in mind for the future.

Encryption

As another small note, data encryption is another way to protect your data.

Certain sensitive data, such as passwords, must always be kept encrypted in the database.

It should be encrypted with a one way hash function.

A security breach means that the passwords of your users will be leaked to the internet.

Users re-use passwords between sites, so if someone's password on your service is leaked, it could very easily be someone's online banking password...

Encryption Keys

With that in mind, encryption keys should also not be stored in the database either... That would make it too easy for attackers!

Or, if they are stored in the database, they can be stored encrypted.

The standard sort of solution to this is to encrypt the key with the user's password.

You don't want to see your company named and shamed on cable TV as having leaked the personal data of your users, do you?