# CSCI 4333 Section 2 Design of DB Systems

## 4/29/2024 (self - annotation)

**DB Security**

by K. Yue

**1. Database Security**

* Protect the database from unauthorized access, modification, or destruction.
* The CIA Model of Security (or AIC Triad)
	1. Confidentiality: accessed only by authorized users.
	2. Integrity: modified only by authorized users.
	3. Availability: accessible when needed.
* Information system *access control* must address:
	1. Authorization: Who have what privileges to which objects?
	2. Identification: E.g., Account names.
	3. Authentication: E.g., Password.
	4. Accountability: E.g., Who have done what actions?
* Some database security mechanisms:
	1. Views: define better access controls.
	2. Security log: journals storing attempted security violations.
	3. Audit trail: Information about SQL operations are stored, such as by using triggers.
	4. Encryption: especially sensitive information such as passwords.
* SQL authorization language:
	1. GRANT statement used for authorization
	2. REVOKE statement used to retract authorization
	3. MySQL directly supports ROLE, which can be used as the basis of a simple Role Based Access Control (RBAC) system. Attribute Based Access Control (ABAC)

***Example:***

CREATE USER 'temp'@'%' IDENTIFIED VIA mysql\_native\_password USING ....;
GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, INDEX, ALTER, SHOW DATABASES, CREATE TEMPORARY TABLES ON \*.\*
     TO 'temp'@'%' REQUIRE NONE WITH MAX\_QUERIES\_PER\_HOUR 0 MAX\_CONNECTIONS\_PER\_HOUR 0 MAX\_UPDATES\_PER\_HOUR 0 MAX\_USER\_CONNECTIONS 0;

**2. SQL Injection (SQLI)**

* A code injection method that takes advantages of dynamic SQL construction in database-driven Web applications.
* One of the most common Web hacking techniques.
* Originated from *improper filtering of special characters*in the target languages (SQL in this case).
* Attackers enter input through Web forms to modify the intention of the SQL statements in the backend Web applications.

***Example: adapted from the textbook and Wikipedia***

Consider a Web form that accepts user names and passwords:

The back-end page may include unsafe code dynamically constructing a query.

query = "SELECT \* FROM users WHERE name = '" + *username* + "' and password = '" + *password* + "';"

The variables username and password get their values from the users through the Web form through the CGI protocol.

Thus, if the user enters (not considering encryption issues here):

username = yue
password = 1Bkm\*2ce

the variable query will have a value of

"SELECT \* FROM users WHERE name = '*yue*' and password = '*1Bkm\*2ce*';"

The query can be executed to get information about the user 'yue' if the right password is provided.

If someone enters:

username = yue
password = 1Bkm\*2*'*ce

query becomes:

"SELECT \* FROM users WHERE name = 'yue' and password = '1Bkm\*2*'*ce';"

Executing the SQL query statements will result in a SQL syntax error in the server-side program since the single quote character ' is an escape character in SQL with special meanings.

For SQL injection, attacker may enter:

username = yue
password = *' OR '1'='1*

query becomes:

"SELECT \* FROM users WHERE name = 'yue' and password = '*' OR '1'='1*';"

Note that the structure of the SQL statement has been changed. Since the and operator has a higher precedence than the or operator, this is equivalent to:

"SELECT \* FROM users WHERE (name = 'yue' and password = '') *OR* '1'='1';"

The condition in the WHERE clause will always be evaluated to true. The query will bypass the password checking and return information about *all* users, not just the user 'yue'.

For input:

username = yue
password = ' OR '1'='1’; DELETE \* FROM student; --

query becomes essentially:

"SELECT \* FROM users
 WHERE name = 'yue' and password = '*' OR '1'='1';
 DELETE \* FROM student; --*'; "

For input:

username = yue
password = ' OR '1'='1’; DROP TABLE users; SELECT \* FROM account; --

query becomes essentially:

"SELECT \* FROM users
 WHERE name = 'yue' and password = '*' OR '1'='1';
DROP TABLE users;
SELECT \* FROM account; --* '; "

In all of these examples, the*intended structures* of the SQL statements are changed by the attackers.

**2.1 SQLI Mitigation**

[1] Input validation: validate input parameter values, properly escaping special characters.

In the minimum, replace one ' by two ':

username = username.replace("'", "''")
password = password.replace("'", "''")

For input parameters:

username = yue
password = ' OR '1'='1

after filtering query becomes:

"SELECT \* FROM users WHERE name = 'yue' and password = '*''* OR *''*1*''*=*''*1';"

The condition of the Where clause will be false (unless the password is really "*' OR '1'='1*".

* Besides possible SQLI, a Web page without proper input validation may result in syntax or runtime errors and reveals information about the back-end system.
* Input validation not only improves security. It has many other benefits.

[2] Using parameterized queries:

* Parameterized queries are usually *prepared and compiled* beforehand with placeholders for input parameters. The structure of the SQL statement cannot be changed. E.g., in Python, %s is a parameter placeholder.

query = "SELECT \* FROM users WHERE name = %s and password = %s;"
cursor.execute(query,(username, password))

* Prepared statements provide many benefits and should always be considered as the first choice.

[3] Using intermediate mid-tiered objects instead of SQL for centralized checking.

* Example: instead of directly executing SQL statements, the Web applications can call methods of well-designed and well-tested classes to access the data.

[4] Use database security features and good practices

* Can be vendor specific.
* Apply the CIA principle to set up the database.

**Introduction to Physical Database**

by K. Yue

**1. Introduction**

* Data in databases is stored in storage medium to provide persistence. Two major questions?
	1. What are the storage media?
	2. How are the data stored?
* I/O operations are usually the most significant factor for database operation latency, not CPU or memory operations.
* The [memory hierarchy](http://en.wikipedia.org/wiki/Memory_hierarchy) distinguishes each level of computer storage by access time. Higher level memory is usually:
	1. faster in access time,
	2. lower in volume (size),
	3. more expensive in cost, and
	4. likely to be not persistent: volatile memory.
* Relational model: tuples and relations -> File systems: records and files.
* Principle of locality: Things are not distributed randomly. They are concentrated in some local areas.
	1. More frequently used data should be stored in memory of higher level in the memory hierarchy.
	2. Data frequently accessed together should be stored close to each other in the storage device, if applicable.
	3. Caching may be used to enhance performance.
* Considerations of selecting secondary storage devices:
	1. Speed
	2. Volume
	3. Cost
	4. Reliability
	5. Availability
* Each type of storage devices has important characteristics that should be considered carefully for uses in database architecture.

***Example***:

An article on [flash memory and database](http://www.hansolav.net/blog/content/binary/HowFlashMemory.pdf) lists important characteristics of flash memory for database consideration:

1. No in-place update: to rewrite a block, one or more sectors may need to be erased and rewritten. Erasion may take time in the mini-second ranges.
2. No mechanical latency: basic operation in micro-seconds, not mini-seconds.
3. Limited number of writes before wearing down.
4. Asymmetric read/write: write operations may be twice as long as read operations.

**2. Physical Database Design**

* Block:
	+ A row (tuple) can be a record in a file system.
	+ To improve performance, records can be grouped and stored in *blocks* to maximize the use of the seek operation in a hard disk.
	+ Blocking factor: the number of records in a block.
	+ Block size: the number of Bytes in the block in one read operation.
* Files can be sequential files, direct access files, or others.
* Examples of important DB file structures:
	+ B+-tree: the *primary key* is used to navigate through an tree index structure to reach linked terminal nodes that store records.
		1. Fast sequential access through the primary key: O(n/B) read operation, where n is the number of records, and B is the blocking factor.
		2. Fast random access through the primary key: O(lg (n/B)).
	+ Hashing: an address is computed from the primary key for storage:
		1. Fast random access through the primary key: O(1).
		2. Slow sequential access through the primary key.
* DBMS may allow you to select the physical structures, sometimes known as the *storage engines*.

**3. Denormalization, Partitioning and Clustering**

* Denormalization: combining tables into one table for faster access.
* Partitioning: breaking down tables for faster access (as tables are smaller).
	+ Horizontal partitioning: distributing rows to component tables.
	+ Vertical partitioning: distributing columns to component tables.
* Clustering: related records from different tables can be stored together in the same disk area.

**4. MySQL Indexes**

* An *index* is an access path created to search for records (tuples) more efficiently.
* There is a cost in creating and maintaining an index.
* Cost effectiveness analysis, including profiling, may be used for consideration of index creation.
* In MySQL, indexes can be created when a table is created: [syntax](http://dev.mysql.com/doc/refman/5.6/en/create-table.html).
* An index creates a physical structure to speed up searching with the indexed attributes.
	1. Benefits: faster search
	2. Costs: maintaining the index structure