Module 9: SQL-Injection Attacks and Multi-Level Database Security

Dr. Natarajan Meghanathan
Associate Professor of Computer Science
Jackson State University
E-mail: natarajan.meghanathan@jsums.edu

SQL Injection Attacks

- SQL injection attack is a code injection technique that exploits the security vulnerabilities in a database application.
- The security vulnerabilities can occur if user input is not filtered for escape characters and/ or if user input is not strongly typed.
- More generally, the vulnerabilities can occur whenever one programming or scripting language is embedded insider another.
- SQL injection attacks are easy to perform as well as easy to avoid.

Statement = “SELECT * FROM ‘CustomerDB’ WHERE ‘name’ = ‘ ”+ userName + “ ‘ AND ‘password’ = ‘ ” + passwd + “ ‘ ; ”

- If the above SQL query was properly executed by passing name to be *natjsu* and password to be *jsunat*, as stored in the database, the SQL statement would become and get executed properly.

Statement = “SELECT * FROM ‘CustomerDB’ WHERE ‘name’ = ‘natjsu‘ AND ‘password’ = ‘jsunat’; ”

SQL Injection Attacks

- One need not know the username nor the password to launch the SQL injection attack. The value passed for name could be *'OR '1'='1* and the value passed for password could be *'OR '1'='1*

Statement = "SELECT * FROM 'CustomerDB' WHERE 'name' = ' "+ userName + " ' AND 'password' = ' " + passwd + " ' ; "

Statement = "SELECT * FROM 'CustomerDB' WHERE 'name' = ' ' OR '1'='1' AND 'password' = ' ' OR '1'='1'; "

- All the records from the Customer Database would be listed.

SQL Injection Attacks

- Another variation using the "--" comment operator:
- One could append the comment "--" operator along with the String for the username and totally avoid executing the password segment of the SQL query. Everything after the -- operator would be considered as comment and not executed.
- To launch such an attack, the value passed for name could be *'OR '1'='1' ; --*

Statement = "SELECT * FROM 'CustomerDB' WHERE 'name' = ' "+ userName + " ' AND 'password' = ' " + passwd + " ' ; "

Statement = "SELECT * FROM 'CustomerDB' WHERE 'name' = ' ' OR '1'='1';-- + " ' AND 'password' = ' " + passwd + " ' ; "

- All the records from the Customer Database would be listed.

SQL Injection Attacks

- Yet, another variation of the SQL Injection Attack can be conducted in DBMS systems that allow multiple SQL statements. Here, we can also make use of the vulnerability in certain DBMS wherein a user supplied field is not strongly typed or is not checked for type constraints.
- This could take place when a numeric field is to be used in a SQL statement; but, the programmer makes no checks to validate that the user supplied input is numeric.
- In the example below if the user passed the string input *1; DROP TABLE 'Users'* as the value for the variable userID, which is supposed to take only Integer input for proper execution, the Users table could be deleted from the DBMS.

Statement = "SELECT * FROM 'CustomerDB' WHERE 'id' = " + userID +";"

Statement = "SELECT * FROM 'CustomerDB' WHERE 'id' = 1; DROP TABLE 'Users' ;

Solutions to Prevent SQL Injection

- Bottom line: User input must not be directly embedded in SQL statements and executed in DBMS programs.
- Solution # 1 (Preferred): Use Parameterized statements: SQL statements can have parameters that act as place holders to which user input is then assigned (bound) at run-time. The following example (adapted from Wikipedia) is using Java and the JDBC API:

```
PreparedStatement prepSt = DBConn.prepareStatement(" SELECT * FROM
'CustomerDB' WHERE username=? AND password=?");

prepSt.setString(1, username);
prepSt.setString(2, password);
prepSt.executeQuery( );
```

Note: Words appearing in light blue color denote built-in JDBC classes and SQL query key words; Words appearing in green color denote built-in functions of the JDBC classes.

Solutions to Prevent SQL Injection

- **Solution # 2: Do not allow multiple SQL queries in a single statement**
- (This is only a partial solution as we have seen in our examples that injection attacks could be launched with only one SQL query through carefully framed user inputs) – Databases like MySQL do not allow multiple SQL queries to be executed as a single statement.

- **Solution # 3 (More Stricter – not flexible): Sanitizing the Input**
- This solution involves limiting the number and the domain of the characters that are acceptable as input. For message boxes in forums, it may be a difficult choice to find out which characters should not be allowed. But, for username and password-kind of smaller fields that involve only certain types of characters and values, this method will often work.

Inference Problem

Lets say we want to find the salary of Laura Richards. A direct query asking the salary of Laura Richards will be considered sensitive and rejected by the DBMS.

On the other hand, we can first list all the female employees in the company and the state they come from. We can then pose a query asking the average salary of all female employees coming from the state of 'MI'. Since Laura Richards will be the only female employee from MI, the average salary returned from the second Query would indeed be the actual salary of Laura Richards.

Select Name, State from Employee where Sex = 'F'

- Sheila Smith IL
- Laura Richards MI
- Michelle Donald CA

Select Average(Salary) from Employee where Sex = 'F' and State = 'MI'

> $65,000

Inference Attack!! – Two queries considered to reveal insensitive data when put together can be used to extract a sensitive information.

Probable Solution: Do not return results on sensitive information that were executed on a single row

Inference Problem –Example 2

- Student Table

Name	Sex	Race	Aid	Fines	Drugs	Dorm
Adams	M	C	5000	45.	1	Holmes
Bailey	M	B	0	0.	0	Grey
Chin	F	A	3000	20.	0	West
Dewitt	M	B	1000	35.	3	Grey
Earhart	F	C	2000	95.	1	Holmes
Fein	F	C	1000	15.	0	West
Groff	M	C	4000	0.	3	West
Hill	F	B	5000	10.	2	Holmes
Koch	F	C	0	0.	1	West
Liu	F	A	0	10.	2	Grey
Majors	M	C	2000	0.	2	Grey

Inference Problem – Example 2

- Inference using SUM:
- We query the database to *report the total of student aid by sex and dorm*.

	Holmes	Grey	West	Total
M	5000	3000	4000	12000
F	7000	0	4000	11000
Total	12000	3000	8000	23000

- The above seemingly innocent report reveals that no female in the GREY dorm is receiving financial aid.
- One can then infer female students like Liu living in GREY do not have financial aid.

Inference Problem – Example 2

- Inference using COUNT and SUM:
  - We query the database to *report the count of students by dorm and sex*

	Holmes	Grey	West	Total
M	1	3	1	5
F	2	1	3	6
Total	3	4	4	11

  - We query the database to *report the total of student aid by sex and dorm*

	Holmes	Grey	West	Total
M	5000	3000	4000	12000
F	7000	0	4000	11000
Total	12000	3000	8000	23000

  - One male student in the Holmes dorm and one male student in the West dorm are receiving financial aid of $5000 and $4000 respectively.
  - Querying the database to report the names of male students in dorm Holmes and in dorm West will help us to figure out that Adams is receiving $5000 as financial aid and Groff is receiving $4000 as financial aid. We cannot blame this particular query to report the name because it is the least sensitive among the three queries used in this indirect attack.

Use of SQL ‘View’

- A View is a virtual table (with rows and columns like a real table), created based on the result-set of an SQL statement (preferably run on a real table).
- SQL CREATE VIEW Syntax
  - CREATE VIEW view_name AS SELECT column_name(s)
    FROM table_name WHERE condition
- For the employee database shown in the previous slide, the DBA could use the following SQL query to create a View per username in the database.

CREATE VIEW Employee_View AS
SELECT * FROM Employee WHERE Name = User;

CREATE USER Bob Kelly identified by bobk; -- creates a user switch for Bob Kelly with a password called bobk.

Likewise a user switch and corresponding View can be created for every user with an entry in the Employee table.

Solving the Inference Problem using ‘Row Level Security’ through SQL View

- Row level security restricts user access at the row (or record) level.
- The idea behind row level security is to allow a user to only access data that pertains or is relevant to the particular user.
- For example, a college student accessing a database to register for classes should only be able to access the data that pertains to himself or herself.
- One solution to the Inference problem for relational databases is to enforce ‘Row Level Security’ implemented through SQL Views.
- The idea is to statically create a View for every valid user of the database and whenever a user logs in and attempts to access the database, he/she can work on only his/her View created for the database. All updates done through the View are replicated in the original database.
- Constraints: A user is restricted to work only on his/her rows (View) and cannot query for any aggregate or statistical information on the database.
- Note that a DBA admin while creating the View for each user could impose constraints on which columns are changeable/ read-only.

Virtual Private Database (VPD)

- Scalability Problem with SQL 'View': If the Employee table had 1000 employee records and suppose we want employees to access their own records only, we will need to create 1000 views.
- The idea behind Virtual Private Database (VPD) is a server-side solution that invokes a policy function that returns a predicate (based on session attributes or application context) and dynamically rewrite the submitted query of the user by appending the returned predicate to the WHERE clause. The modified SQL query is then executed.
- The idea of Virtual Private Database has been implemented in Oracle.
- A simple policy function could be the one that returns a String depending on the username for the login session.
  - If the username corresponds to the DBA, then an empty String (predicate) is returned and the SQL query is executed without appending the predicate to the WHERE clause.
  - If the username is not the DBA, then a String corresponding to the username is returned as the predicate and the SQL query is executed by appending the predicate to the WHERE clause.

Example for Virtual Private Database

Consider the Employee table shown previously:

If the Username is ‘Bob Kelly’, the DBA Employee, no predicate would be appended to the following SQL query and would be executed as it is:

Select Average(Salary) from Employee where Sex = ‘F’
> $60,000;

If the Username is ‘Alex Ryder’ - a non-DBA Employee, then his username would be appended to the SQL query and executed as:

Select Average(Salary) from Employee where Sex = ‘F’ AND Name = ‘Alex Ryder’
➢NULL; -- implying no matching record

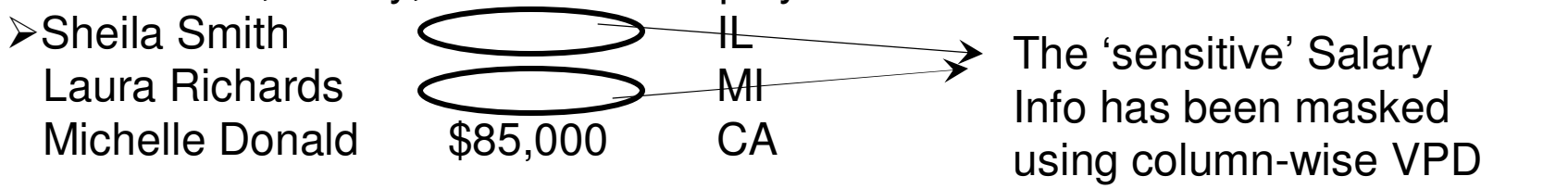
If the Username is ‘Michelle Donald’ - a non-DBA Employee, then her username would be appended to the SQL query and executed as:

Select Average(Salary) from Employee where Sex = ‘F’ AND Name = Michelle Donald’
> $85,000;

Column-wise Masking VPD

- The previous slides described for VPD represent what is called “Row-level VPD,” because it restricts the number of rows returned from a SQL query to 1. It has the same restriction (of Views) that a user cannot place any queries for statistical or aggregate information.
- With Column-wise VPD, the DBA can declare what columns (i.e., attributes) of the database table are sensitive and only those columns are not considered while executing the query on rows not corresponding to the user. Note that all columns would be considered while executing a SQL query on the row corresponding to the user.
- If the ‘Salary’ column has been declared as the sensitive column in a VPD policy function, then the following query executed by user Michelle Donald would return the rows as shown below.

Select Name, Salary, State from Employee where Sex = ‘F’



Thus, a column-wise VPD can fetch/ execute records from multiple rows!!!

Multi-Level Security (MLS) Database

- With the use of Multi-Level Security (MLS), a DBMS can allow subjects with different security clearances to simultaneously access objects with different security levels.
- The Security clearances and security levels typically considered are: Top Secret (TS), Secret (S), Confidential (C) and Unclassified (U).
- MLS allows subjects with higher security clearance to easily allow access objects with equal or lower security level.
- The ordering among these clearances and security levels is as follows:
  - TS > S > C > U
- The MLS approach is a classical example for the Mandatory Access Control (MAC) model as the security clearance and security level for the subjects and objects are uniformly adopted enterprise-wide.
- Typically, each field (element object) in the table is assigned a security level and the security level for a tuple is the highest of the security levels of its constituent element objects.

Example for MLS Database

- Consider the following Storage database wherein, for simplicity, each tuple is assigned a security level.

Room No.	Compartment	Description	Security Level
450	X	Textbooks	Unclassified
450	Y	Computers	Unclassified
451	X	Bank Records	Confidential
451	Y	Furniture	Unclassified
452	X	Food	Unclassified
452	Y	Research Equipment	Unclassified

- Each room contains two compartments (X and Y). The combination of the room number and the compartment forms the Primary Key.
- There has to be “two” views of the above table. The DBA and other managerial-level users have to be able to see all the six tuples of the above table; whereas, a regular user should not be able to see the third tuple that has a Confidential information (Bank Records!!)

Example for MLS Database

- If a regular user with security clearance ‘Unclassified’ runs the following SQL Query
  - Select * from Storage
- The results should be:

Room No.	Compartment	Description	Security Level
450	X	Textbooks	Unclassified
450	Y	Computers	Unclassified
451	Y	Furniture	Unclassified
452	X	Food	Unclassified
452	Y	Research Equipment	Unclassified

- However, the above display of records leads the Unclassified user to believe that there is nothing in the Compartment X of Room no. 451.
- If the user executes an Insert SQL query like “INSERT INTO Storage VALUES (‘451’, ‘X’, ‘Fruits’, ‘Unclassified’); then the DBMS would return an Access denied feedback.

Example for MLS Database

- The ‘Access Denied’ feedback could mean to the Unclassified user that there is already a tuple with the primary key matching with what he is trying to insert and that the security level of the tuple is greater than his security clearance. This is referred to as an “Indirect Channel.”
- If the employee comes to know about the presence of something that is more than “Unclassified” in Compartment X of Room no. 451, then he/she could potentially do something harmful to the entity being stored over there and cause damage to the organization.
- Polyinstantiation
- To avoid such a dangerous inference, the organization could allow the employee to insert a tuple at the Unclassified level in the database. However, this would lead to a situation wherein there are two tuples with the same value for the primary key: Room no. 451 and Compartment X.
- The presence of two or more rows (tuples) with the same value(s) for the primary key, obviously with different security levels for each tuple is called “Polyinstantiation.”

Example for MLS Database Polyinstantiation

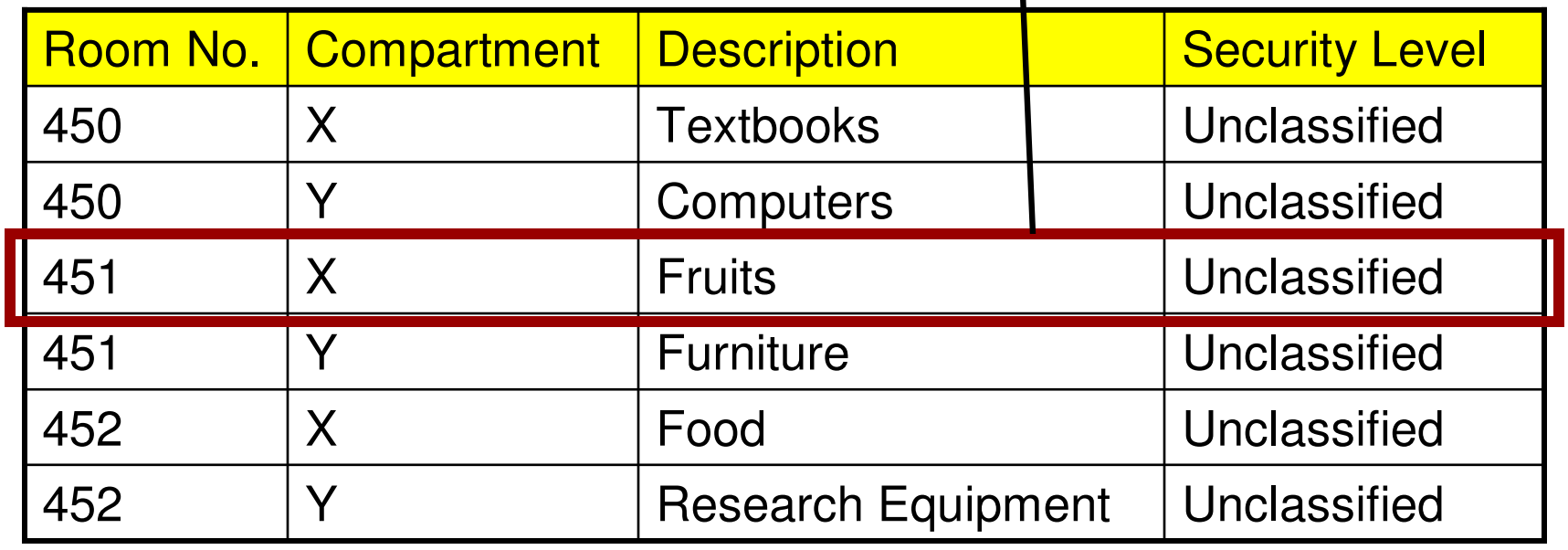
Room No.	Compartment	Description	Security Level
450	X	Textbooks	Unclassified
450	Y	Computers	Unclassified
451	X	Bank Records	Confidential
451	X	Fruits	Unclassified
451	Y	Furniture	Unclassified
452	X	Food	Unclassified
452	Y	Research Equipment	Unclassified

Polyinstantiated Rows

- When user with security classification “Confidential” or above executes the SQL query “Select * from Storage,” all the above 7 tuples would be returned. The user has to however discard the polyinstantiated tuples with security level below to that of the user.

Example for MLS Database Polyinstantiation

- When user with security classification “Unclassified” executes the SQL query “Select * from Storage,” the tuples that are of security level “Unclassified” would only be returned. The user is made to believe that there are Fruits in Compartment X of Room No. 451. This is better than displaying “NULL” for the sensitive attribute and raising suspicion for the Unclassified that something sensitive is over there.
- For the organization, this will appear like a “**Cover Story**” that it spreads among users who are not authorized to know the actual information pertaining to that tuple.



Room No.	Compartment	Description	Security Level
450	X	Textbooks	Unclassified
450	Y	Computers	Unclassified
451	X	Fruits	Unclassified
451	Y	Furniture	Unclassified
452	X	Food	Unclassified
452	Y	Research Equipment	Unclassified

Visible and Invisible Polyinstantiation

- Visible Polyinstantiation occurs when a user with a higher security clearance attempts to insert a tuple into a database that already has a tuple, with the same primary key, at a lower security level.

- Invisible Polyinstantiation occurs when a user with a lower security clearance attempts to insert a tuple into a database that already has a tuple, with the same primary key, at a higher security level.
- The Polyinstantiation that we created in the previous slides (Storage database Example) is an example for Invisible Polyinstantiation.
- Consequence of Polyinstantiation:
- An after-effect of allowing polyinstantiation is that there will be an explosion of tuples in a database. If the number of security levels is 4, for every tuple, that could be entered with the highest security level in the database table, there would be at most three “Cover Story” tuples that could be recorded.
- Polyinstantiation is inevitable in Multi-level world!!