Database Security

Database Systems

Michael Pound

This Lecture

- General Database Security
- Privileges
  - Granting
  - Revoking
- Views
- SQL Insertion Attacks
- Further Reading
  - The Manga Guide to Databases, Chapter 5
  - Database Systems, Chapter 20

Database Security

- Database security is about controlling access to information
  - Some information should be available freely
  - Other information should only be available to certain people or groups
- Many aspects to consider for security
  - Physical security
  - OS/Network security
  - Encryption and passwords
  - DBMS security
- This lecture we will focus mainly on DBMS security

DBMS Security Support

- DBMSs can provide some security
  - Each user has an account, username and password
  - These are used to identify a user and control their access to information
- The DBMS verifies password and checks a user's permissions when they try to
  - Retrieve data
  - Modify data
  - Modify the database structure

Permissions and Privilege

- SQL uses privileges to control access to tables and other database objects. E.g.
  - SELECT privilege
  - INSERT privilege
  - UPDATE privilege
  - CREATE privilege
- In MySQL there are actually 30 distinct privileges
- The owner (creator) of a database has all privileges on all objects in the database, and can grant these to others
- The owner (creator) of an object has all privileges on that object and can pass them on to others

Privileges in SQL

```
GRANT <privileges>
 ON <object>
 TO <users>
[WITH GRANT OPTION]
```

<privileges> is a list of **SELECT (<columns>)**, **INSERT (<columns>)**, **DELETE**, **UPDATE (<columns>)**, or simply **ALL**

- **<users>** is a list of user
- **<object>** is the name of a table or view
- **WITH GRANT OPTION** means that the users can pass their privileges on to others

Privileges Examples

```
GRANT ALL ON
 Employee
 TO Manager
 WITH GRANT OPTION;
```

- The user 'Manager' can do anything to the Employee table, and can allow other users to do the same (by using GRANT statements)

```
GRANT SELECT,
 UPDATE(Salary)
 ON Employee
 TO Finance;
```

- The user 'Finance' can view the entire Employee table, and can change Salary values, but cannot change other values or pass on their privilege

Removing Privileges

- If you want to remove a privilege you have granted you use

```
REVOKE
 <privileges>
 ON <object>
 FROM <users>;
```

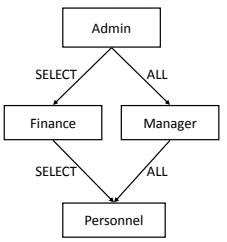
- For example:

```
REVOKE
 UPDATE(Salary)
 ON Employee
 FROM Finance

REVOKE ALL
 PRIVILEGES, GRANT
 OPTION FROM
 Manager
```

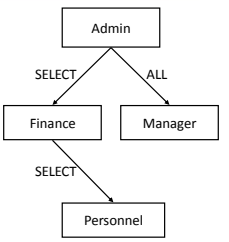
Removing Privileges

- Example
  - 'Admin' grants ALL privileges to 'Manager', and SELECT to 'Finance' with grant option
  - 'Manager' grants ALL to Personnel
  - 'Finance' grants SELECT to Personnel



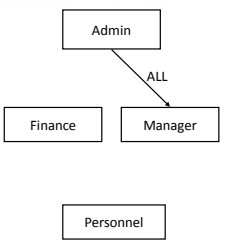
Removing Privileges

- Manager' revokes ALL from 'Personnel'
  - 'Personnel' still has SELECT privileges from 'Finance'



Removing Privileges

- Manager' revokes ALL from 'Personnel'
  - 'Personnel' still has SELECT privileges from 'Finance'
- 'Admin' revokes SELECT from 'Finance'
  - Personnel also loses SELECT



Views

- Privileges work at the level of tables
  - You can restrict access by column
  - You cannot restrict access by row
- Views, along with privileges, allow for customised access
- Views provide 'virtual' tables
  - A view is the result of a SELECT statement which is treated like a table
  - You can SELECT from (and sometimes UPDATE etc) views just like tables

Creating Views

```
CREATE VIEW <name>
 AS
<select statement>;
```

- <name> is the name of the new view
- <select statement> is a query that returns the rows and columns of the view

- Example
  - We want each university tutor to be able to see marks of only those students they actually teach
  - We will assume our database is structured with Student, Enrolment, Tutors and Module tables similar to those seen in previous lectures

View Example

Student

sID	sFirst	sLast	sYear

Enrolment

sID	mCode	eMark	eYearTaken

Module

mCode	mTitle	mCredits

Tutors

lID	sID

Lecturers

lID	lName	lDept

View Example

```
CREATE VIEW TuteeMarks
AS
SELECT sID, sFirst, sLast, mCode, eMark
FROM Student INNER JOIN Enrolment USING(sID)
             INNER JOIN Module USING (mCode)
WHERE sID IN (SELECT sID FROM Tutors
              WHERE lID = CURRENT_USER);

GRANT SELECT ON TuteeMarks TO 'user'@'%';
```

Note: You should grant for all Tutors in MySQL, in Oracle you can grant to PUBLIC. In Oracle CURRENT_USER is called USER

Database Integrity

- Database Security
  - Database security makes sure that the user is authorised to access information
  - Beyond security, checks should be made that user mistakes are detected and prevented
- Database Integrity
  - Ensures that authorised users only input consistent data into the database
  - Usually consists of a series of constraints and assertions on data

Database Integrity

- Integrity constraints come in a number of forms:
  - CREATE DOMAIN can be used to create custom types with specific values
  - CREATE ASSERTION can be used to check manipulation of tables against some test, that must always be true
  - CHECK constraints (more widely suppoted) are used to check row-level constraints
- Oracle supports CHECK constraints. MySQL can emulate them with triggers

Connections to a DBMS

- A major concern with database security should be when your application connects to the DBMS
  - The user doesn't connect to the DBMS, the application does
  - This often happens with elevated privileges
  - If the application isn't well secured, it could provide a conduit for malicious code

SQL Injection Attacks

An SQL Injection attack is an exploit where a user is able to insert malicious code into an SQL query, resulting in an entirely new query

SQL Injection Attacks

- It is common for user input to be read, and form part of an SQL query. For example, in PHP:

```
$query = "SELECT * FROM Products
 WHERE pName LIKE '%" . $searchterm . "%'";
```

- If a user is able to pass the application malicious information, this information may be combined with regular SQL queries
- The resulting query may have a very different effect

SQL Injection Attacks

- An application or website is vulnerable to an injection attack if the programmer hasn't added code to check for special characters in the input:
  - ' represents the beginning or end of a string
  - ; represents the end of a command
  - /*...*/ represent comments
  - -- represents a comment for the rest of a line

SQL Injection Attacks

- Imagine a user login webpage that requests a user ID and password. These are passed from a form to PHP via $_POST
  - $_POST['id'] = 'Michael'
  - $_POST['pass'] = 'password'
- The ID is later used for a query:

```
SELECT uPass FROM Users
 WHERE uID = 'Michael';
```

SQL Injection Attacks

- In PHP the code for any user might look something like this:

```
$query = "SELECT uPass FROM Users WHERE
  uID = '" . $_POST['id'] . "'";
$result = mysql_query($query);
$row = mysql_fetch_row($result);
$pass = row['uPass'];
```

- The password would then be compared with the other field the user entered

SQL Injection Attacks

- If the user enters *Name*, the command becomes:

```
SELECT uPass FROM Users
 WHERE uID = 'Name';
```

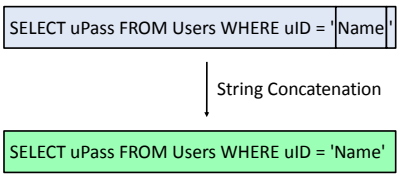
- But what about if the user entered

```
';DROP TABLE Users;--
```

as their name?

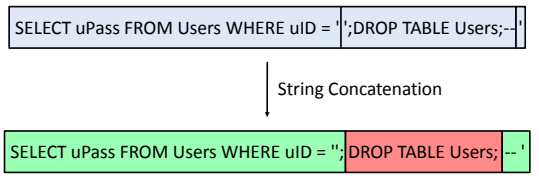
SQL Injection Attacks

- The website programmer intended to execute a single SQL query:



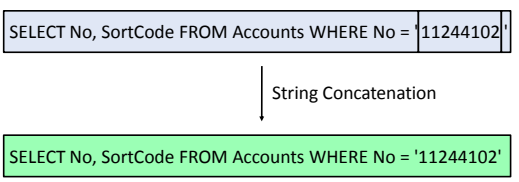
SQL Injection Attacks

- With the malicious code inserted, the meaning of the SQL changes into two queries and a comment:



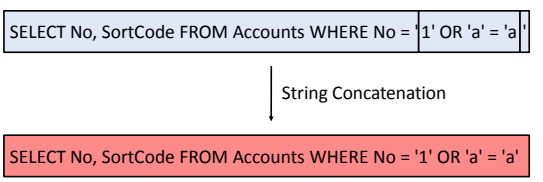
SQL Injection Attacks

- Sometimes the goal isn't sabotage, but information
- Consider an online banking system:



SQL Injection Attacks

- This attack is aimed at listing all accounts at a bank. The SQL becomes a single, altered query:



This is particularly effective with weakly typed languages like PHP

How To Write An SQL Injection Attack

- Data Protection Act 1998, Section 55(1):

  A person must not knowingly or recklessly, without the consent of the data controller obtain or disclose personal data or the information contained in personal data.

- Do not do this on a website you do not own
- *"I was just seeing if it would work"* is not a valid defence

Defending Against Injection Attacks

- Defending against SQL injection attacks is not difficult, but a lot of people still don't
- There are numerous ways you can improve security. You should be doing most of these at any time where a user inputs variables that will be used in an SQL statement
- In essence, don't trust that all users will do what you expect them to do

1. Restrict DBMS Access Privileges

- Assuming an SQL injection attack is successful, a user will have access to tables based on the privileges of the account that the application used to connect to the DBMS
- GRANT an application or website the minimum possible access to the database
- Do not allow DROP, DELETE etc unless absolutely necessary
- Use Views to hide as much as possible

2. Encrypt Sensitive Data

- Storing sensitive data inside your database can always lead to problems with security
- If in doubt, encrypt sensitive information so that if any breaches occur, damage is minimal
- Another reason to encrypt data is the majority of commercial security breaches are inside jobs by trusted employees
- **Never** store unencrypted passwords. Many shops still do this

3. Validate Input

- Arguably the most important consideration when creating a database or application that handles user input
- Filter any escape characters and check the length of the input against expected sizes
- Checking input length should be standard practice. This applies to programming in general, as it also avoids buffer overflow attacks

3. Validate Input

- ***Always*** escape special characters. All languages that execute SQL strings will allow this, in PHP:

```
$username = mysql_real_escape_string($input);
$query = "SELECT * FROM Users
          WHERE uID = '" . $username . "'";
$result = mysql_query($query);
```

- mysql_real_escape_string() will escape any special characters, like ', with \
- You should do this with any input variables

4. Check Input Types

- In weakly typed languages, check that the user is providing you with a type you'd expect
- For example, if you expect the ID to be an int, make sure it is. In PHP:

```
if (!is_int($_POST['userid']))
{
   // ID is not an integer
}
```

5. Stored Procedures

- Some DBMSs allow you to store procedures for use over and over again
- Procedures you might store are SELECTs, INSERTSs etc, or other procedural code
- This adds another level of abstraction between the user and the tables
- If necessary, a stored procedure can access tables that are restricted to the rest of the application

6. Generic Error Messages

- While it might seem helpful to output informative error messages, this actually supplies users with far too much information
- For example, if your SQL query fails, do not show the user mysql_error(), instead output:

  A system error has occured. We apologise for the inconvenience.
- You can log the error privately for administrative purposes

7. Parameterised Input

- Parameterised input essentially means that user input is passed to the database as parameters, not as part of the SQL string
- This makes injection attacks extremely difficult
- Not all DBMSs / Languages support this
- In PHP, you need to use PHP Data Objects (PDO)
- Reference: http://php.net/manual/en/book.pdo.php

PDO

- Rather than building up a string for your SQL and executing it, given a PDO mysql connection $conn:

```
$stmt = $conn->prepare('SELECT * FROM Users
                        WHERE uName = :name');
$stmt->bindValue(':name', $_POST['username']);
$stmt->execute();
```

- The statement is pre-compiled during prepare. While a malicious parameter may still be passed to the query, it is simply used rather than executed.