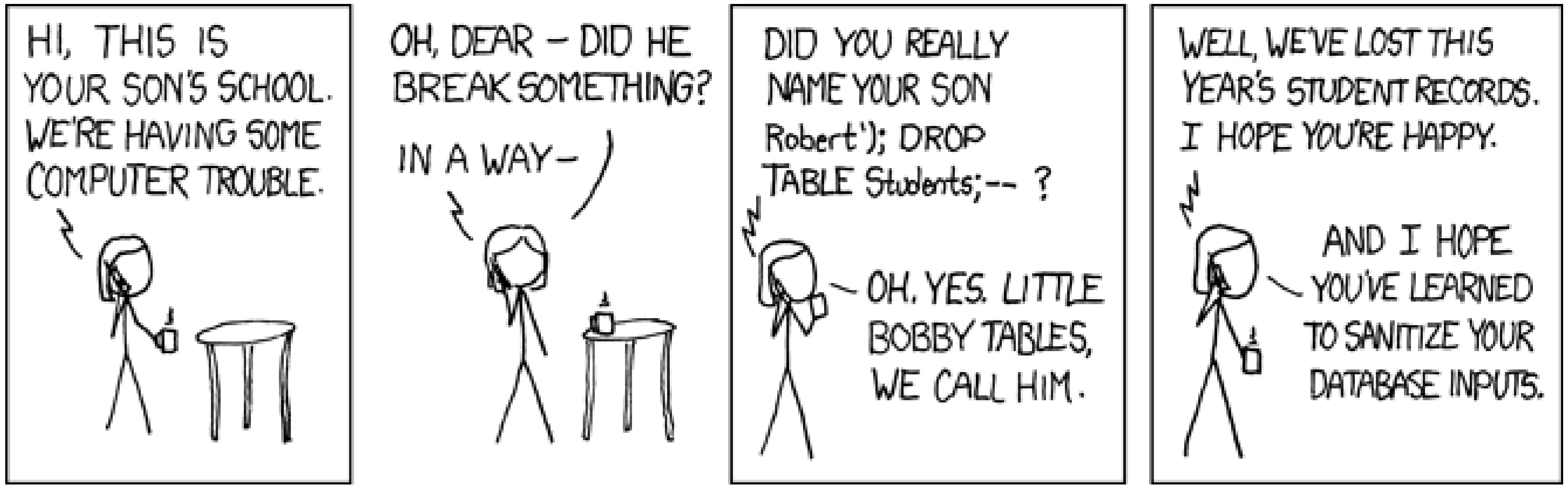
COSC265 – Database Systems

Database Security – SQL Injection

Does this xkcd comic make sense? If not, it will soon…


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Secure Application Development

- Access to Database or Environment Through Applications
- Need to consider security of applications using database as well as security of data in database itself
- Example: SQL Injection Attack

SQL Injection

- SQL Injection
  - Definition – inserting malicious SQL code through an application interface
    - Often through web application, but possible with any interface
  - Typical representative scenario
    - Three-tier application (web interface, application, database)
    - Overall application tracks own usernames and passwords in database (advantage: can manage users in real time)
    - Web interface accepts username and password, passes these to application layer as parameters

SQL Injection (2)

- Example: Application Java code contains a dynamically formed SQL statement:
  - String query = "SELECT * FROM users_table " +
    " WHERE username = " + " ‘ " + username + " ‘ " +
    " AND password = " + " ‘ " + password + " ‘ " ;
- Note: String values must be single quoted in SQL, so application provides this for each passed string parameter
- Expecting one row to be returned if success, access granted
- No rows returned if failure; no access in this case
- Common variant – SELECT COUNT(*) FROM …

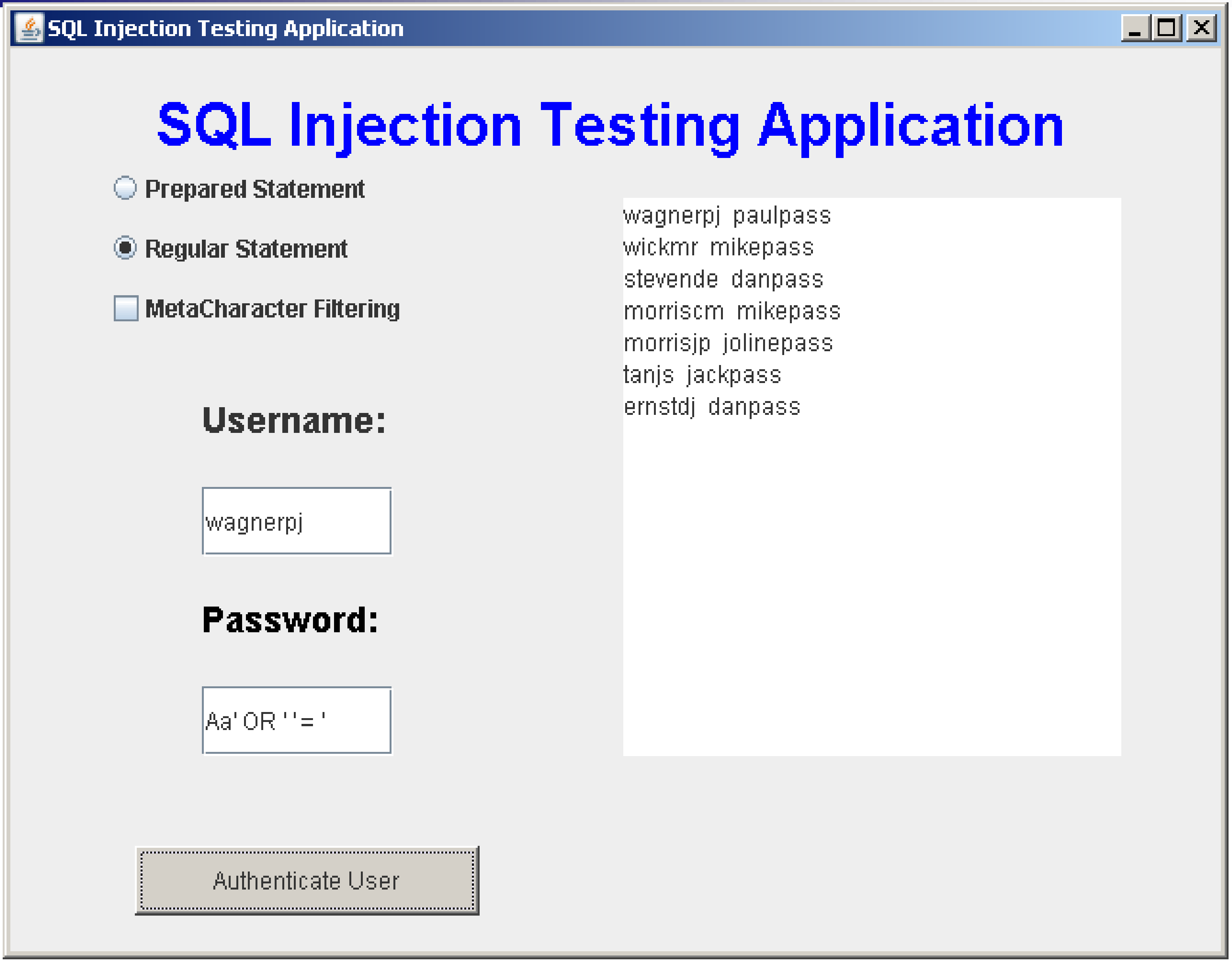
SQL Injection (3)

- Normal (valid) usage:
  - Username: wagnerpj
  - Password: paulpass
  - query =>

```
SELECT *
FROM users_table
WHERE username = ‘wagnerpj‘
AND password = ‘paulpass‘
```

SQL Injection (4)

- However, attacker enters:
  - any username (valid or invalid)
  - password of: Aa’ OR ‘ ‘ = ‘
- Query becomes: SELECT * FROM users_table WHERE username = ‘anyname‘ AND password = ‘Aa‘ OR ‘ ‘ = ‘ ‘
- Note: WHERE clause => F and F or T => F or T => T
  - AND has higher precedence than OR
- All user/pass rows returned to application
- If application checking for 0 vs. more than 0 rows, attacker is in


SQL Injection Testing Application

SQL Injection Testing Application

( ) Prepared Statement

(●) Regular Statement

☐ MetaCharacter Filtering

Username:

wagnerpj

Password:

Aa' OR ''='

Authenticate User

```
wagnerpj  paulpass
wickmr  mikepass
stevende  danpass
morriscm  mikepass
morrisjp  jolinepass
tanjs  jackpass
ernstdj  danpass
```

Ethics/Legal Issues for SQL Injection

- Only try this on:
  - Your own system(s)
  - Systems over which you have explicit permission to do security testing
- It may be considered unethical and/or criminal to probe systems over which you don’t have permission to do so

SQL Injection - Prevention

- What's the problem here?
  - Not checking and controlling input properly
    - Specifically, not controlling string input
  - Note: there are a variety of ways SQL injection can happen
    - Regular inclusion of SQL metacharacters through
      - Variable interpolation
      - String concatenation with variables and/or constants
      - String format functions like sprintf()
      - String templating with variable replacement
    - Hex or Unicode encoded metacharacters

SQL Injection Prevention (2)

- How to resolve this?
  - First (Attempted) Solution: Check Content
    - Client code checks to ensure certain content rules are met
    - Server code checks content as well
    - Specifically – don’t allow apostrophes to be passed
    - Problem: there are other characters that can cause problems; e.g.
      - `--`   // SQL comment character
      - `;`   // SQL command separator
      - `%`   // SQL LIKE subclause wildcard character
    - Which characters do you filter (blacklist) / keep (whitelist)?

SQL Injection – Variant 1

- Any username, password: ‘ or 1=1--
  - Note: -- comments out rest of line, including terminating single quote in application
- Query becomes:
  - SELECT *

    FROM users_table

    WHERE username = ‘anyname‘

    AND password = ‘ ‘ OR 1=1--‘

SQL Injection – Variant 2

- Any username, password: foo’;DELETE FROM users_table WHERE username LIKE ‘%
- Query becomes: SELECT * FROM users_table WHERE username = ‘anyname‘ AND password = ‘foo‘; DELETE FROM users_table WHERE username LIKE ‘%’
- Note: system executes two statements
  - SELECT * FROM users_table WHERE username = ‘anyname’ AND password = ‘foo’ // returns nothing
  - DELETE FROM users_table WHERE username LIKE ‘%’
    - Depending on level of privilege for executing user…

SQL Injection – Prevention (3)

- Review
  - Regular Statements
    - SQL query is generated entirely at run-time
    - Custom procedure and data are compiled and run
      - Compilation allows combination of procedure and data, allowing problems with SQL metacharacters

```
String sqlQuery = null;
Statement stmt = null;
sqlQuery = "select * from users where " +
   "username = " + "'" + fe.getUsername() + "'" + " and " +
   "upassword = " + "'" + fe.getPassword() + "'";
stmt = conn.createStatement();
rset = stmt.executeQuery(sqlQuery);
```

SQL Injection – Prevention(4)

- Better Solution
  - Prepared Statements
    - SQL query is precompiled with placeholders
    - Data is added in at run-time, converted to correct type for the given fields

String sqlQuery = null;

PreparedStatement pStmt = null;

sqlQuery = "select * from users where username = ***?*** and
upassword = ***?***";

pStmt = conn.prepareStatement(sqlQuery);

pStmt.setString(1, fe.getUsername());

pStmt.setString(2, fe.getPassword());

rset = pStmt.executeQuery();

SQL Injection – Prevention (5)

- Issues with PreparedStatements
  - Cannot use them in all situations
    - Generally limited to replacing field values in SELECT, INSERT, UPDATE, DELETE statements
      - E.g. our use for username field value, password field value
    - Example: if also asking user for information that determines choice of table name, cannot use a prepared statement

SQL Injection Prevention (6)

- Additional Precautions
  - Do not access the database as a privileged user
    - Any user who gains access will have that user's privileges
  - Limit database user to only what they need to do
    - e.g. reading information from database, no insert/update/delete
  - Do not allow direct access to database from the internet
    - Require users to go through your applications
  - Do not embed database account passwords in your code
    - Encrypt and store them in a repository that is read at application startup
  - Do not expose information in error messages
    - E.g. do not display stack traces