

CSC 405

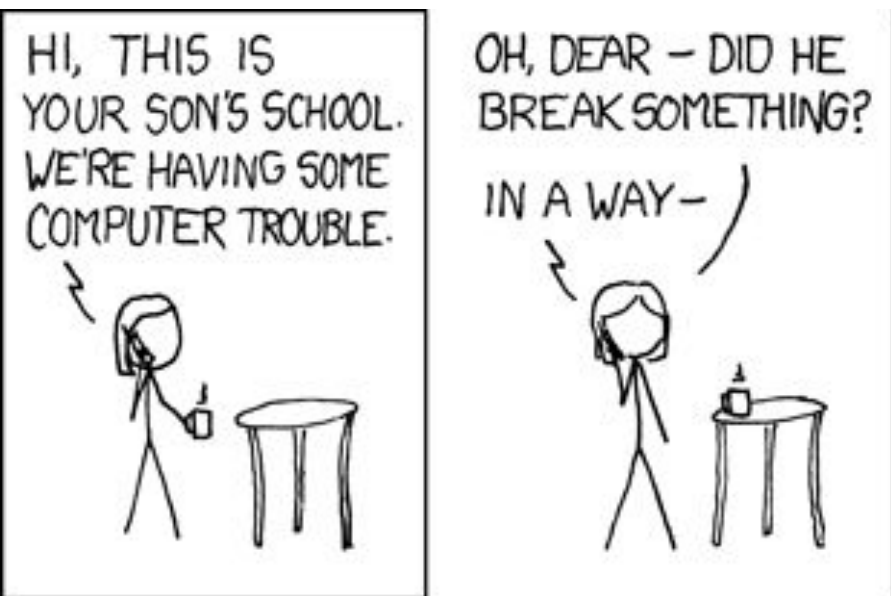
Computer Security

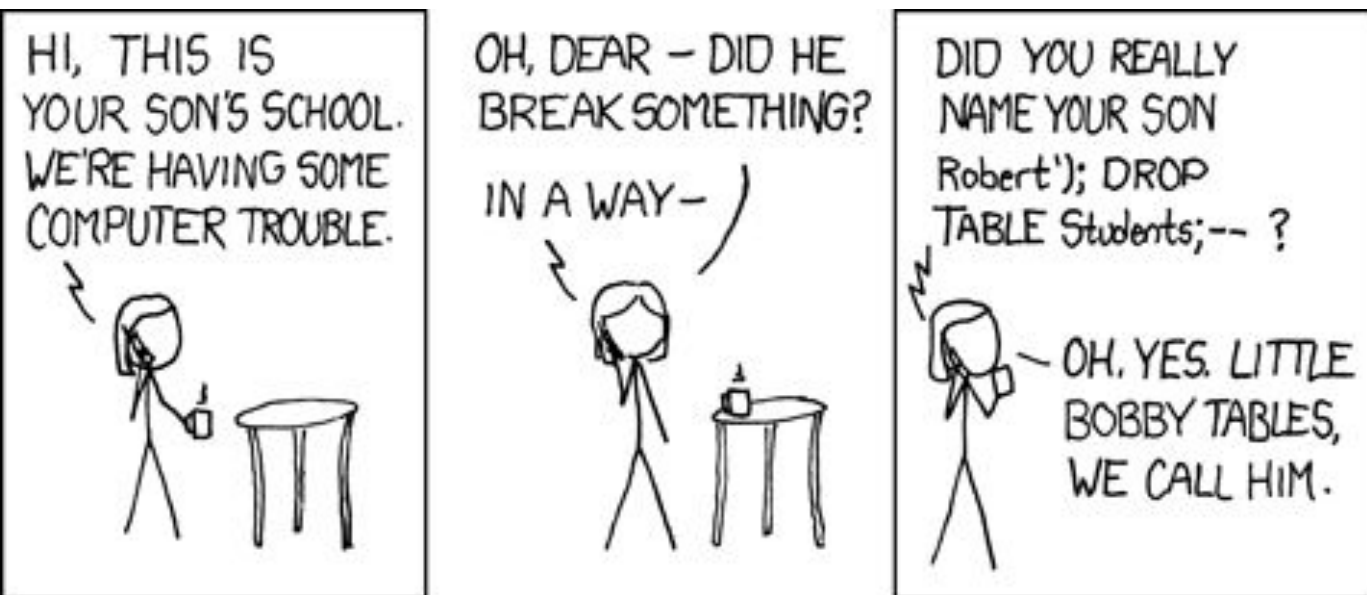
Web Security

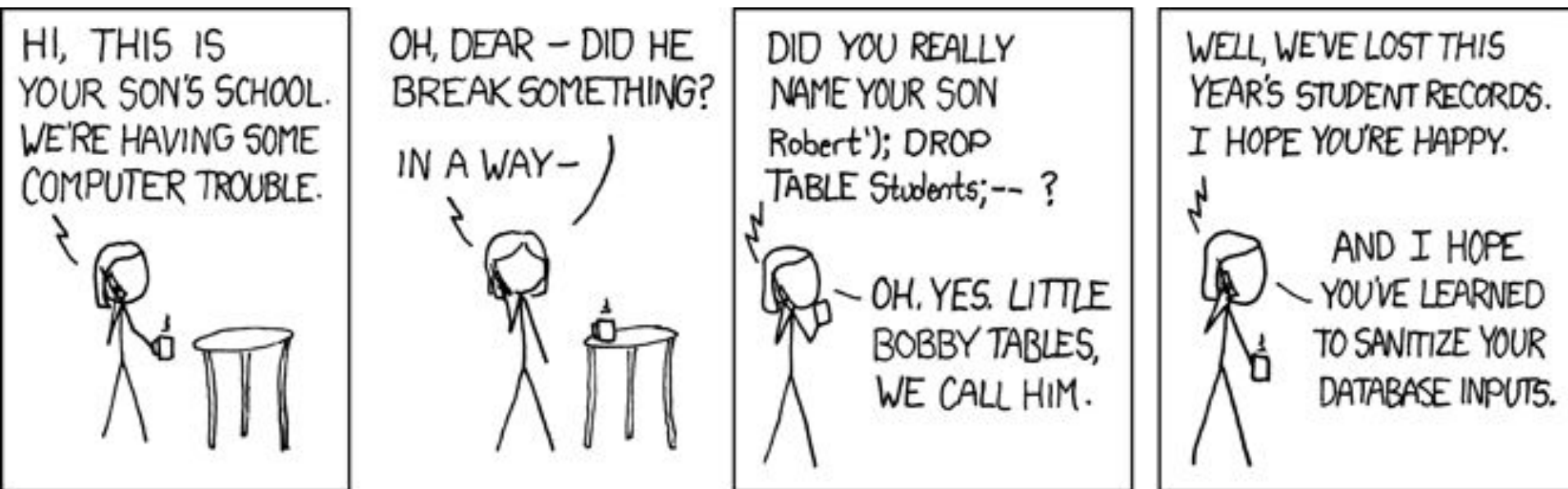
Alexandros Kapravelos
akaprav@ncsu.edu

(Derived from slides by Giovanni Vigna and Adam Doupe)





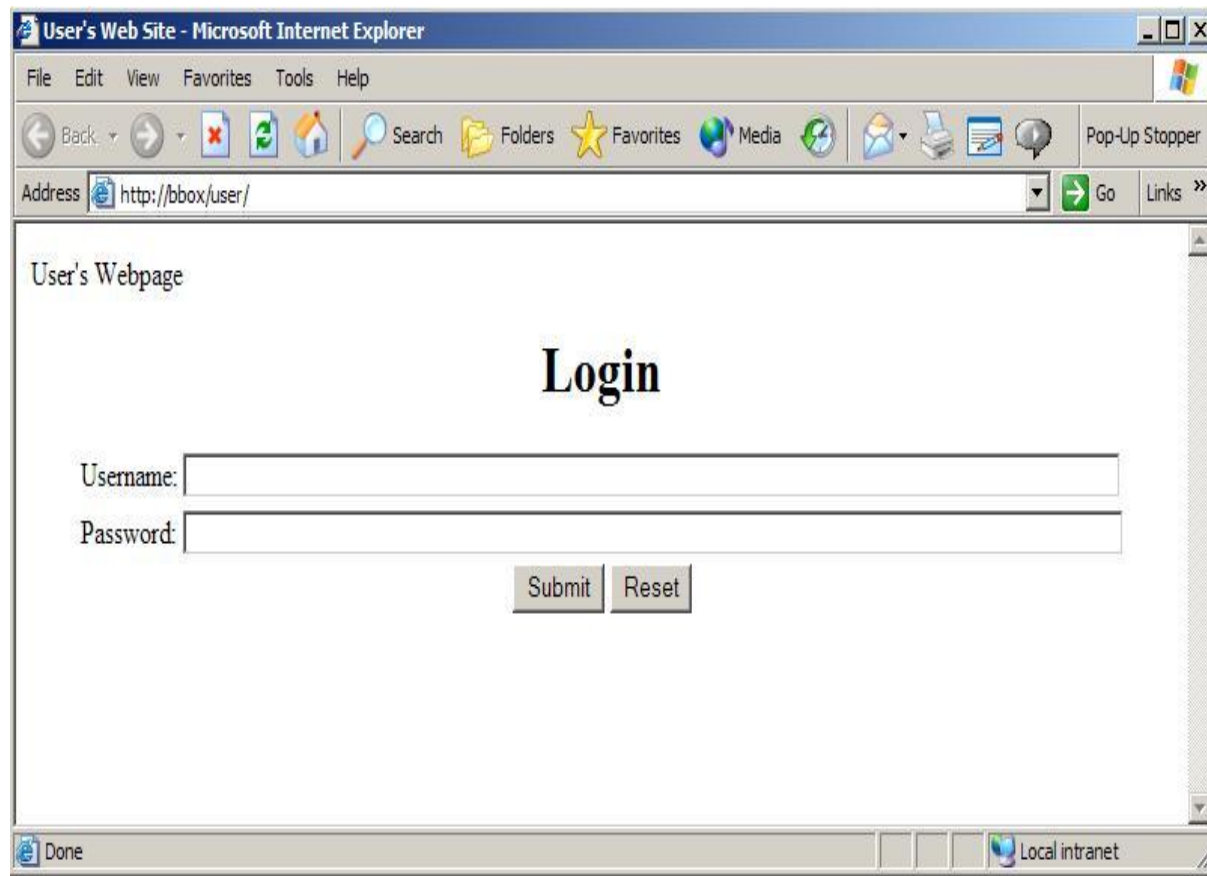




SQL Injection

- SQL injection might happen when queries are built using the parameters provided by the users
 - \$query = "select ssn from employees where name = ' " + username + " ' "
- By using special characters such as ' (tick), -- (comment), + (add), @variable, @@variable (server internal variable), % (wildcard), it is possible to:
 - Modify queries in an unexpected way
 - Probe the database schema and find out about stored procedures
 - Run commands (e.g., using xp_commandshell in MS SQL Server)

An Example Web Page



The Form

```
<form action="login.asp" method="post">
  <table>
    <tr><td>Username:</td>
      <td><input type="text" name="username"></td></tr>
    <tr><td>Password:</td>
      <td><input type="password" name="password"></td></tr>
  </table>
  <input type="submit" value="Submit">
  <input type="reset" value="Reset">
</form>
```


The Login Script

```
... <% function Login( connection ) {  
    var username = Request.form("username");  
    var password = Request.form("password");  
    var rso = Server.CreateObject("ADODB.Recordset");  
    var sql = "select * from pubs.guest.sa_table \  
              where username = '" + username + "' and \  
              password = '" + password + "'";  
    rso.open(sql, connection); //perform query  
    if (rso.EOF) //if record set empty, deny access  
    { rso.close();  
    %>    <center>ACCESS DENIED</center> <%  
    } else { //else grant access  
    %>    <center>ACCESS GRANTED</center> <%  
    // do stuff here ...
```

The ' or 1=1 -- Technique

- Given the SQL query string:

```
"select * from pubs.guest.sa_table \  
  where username = '' + username + '' and \  
  password = '' + password + ''";
```
- By providing the following username:

```
' or 1=1 --
```
- the user name (and any password) results in the string:

```
select * from sa_table where username='' or 1=1 --' and  
password= ''
```

 - The conditional statement “`username='' or 1=1 --`” is true whether or not username is equal to “
 - The “`--`” makes sure that the rest of the SQL statement is interpreted as a comment and therefore `and password =''` is not evaluated

Injecting SQL Into Different Types of Queries

- SQL injection can modify any type of query such as
 - SELECT statements
 - `SELECT * FROM accounts WHERE user='${u}' AND pass='${p}'`
 - INSERT statements
 - `INSERT INTO accounts (user, pass) VALUES('${u}', '${p}')`
 - Note that in this case one must figure out how many values to insert
 - UPDATE statements
 - `UPDATE accounts SET pass='${np}' WHERE user= '${u}' AND pass='${p}'`
 - DELETE statements
 - `DELETE * FROM accounts WHERE user='${u}'`

Identifying SQL Injection

- A SQL injection vulnerability can be identified in different ways
 - Negative approach: special-meaning characters in the query will cause an error (for example: user=" ' ")
 - Positive approach: provide an expression that would NOT cause an error (for example: "17+5" instead of "22", or a string concatenation)

The UNION Operator

- The UNION operator is used to merge the results of two separate queries
- In a SQL injection attack this can be exploited to extract information from the database
- Original query:
 - `SELECT id, name, price FROM products WHERE brand='${b}'`
- Modified query passing `${b}="foo' UNION..."`:
 - `SELECT id, name, price FROM products WHERE brand='foo' UNION SELECT user, pass, NULL FROM accounts -- '`
- For this attack to work the attacker must know
 - The structure of the query (number of parameters and types have to be compatible: NULL can be used if the type is not known)
 - The name of the table and columns

Determining Number and Type of Query Parameters

- The number of columns in a query can be determined using progressively longer NULL columns until the correct query is returned
 - UNION SELECT NULL
 - UNION SELECT NULL, NULL
 - UNION SELECT NULL, NULL, NULL
- The type of columns can be determined using a similar technique
 - For example, to determine the column that has a string type one would execute:
 - UNION SELECT 'foo', NULL, NULL
 - UNION SELECT NULL, 'foo', NULL
 - UNION SELECT NULL, NULL, 'foo'

Determining Table and Column Names

- To determine table and column names one has to rely on techniques that are database-specific
 - Oracle
 - By using the user_objects table one can extract information about the tables created for an application
 - By using the user_tab_column table one can extract the names of the columns associated with a table
 - MS-SQL
 - By using the sysobjects table one can extract information about the tables in the database
 - By using the syscolumns table one can extract the names of the columns associated with a table
 - MySQL
 - By using the information_schema one can extract information about the tables and columns

Second-Order SQL Injection

- In a second-order SQL injection, the code is injected into an application, but the SQL statement is invoked at a later point in time
 - e.g., Guestbook, statistics page, etc.
- Even if application escapes single quotes, second order SQL injection might be possible
 - Attacker sets user name to: `john'--`, application safely escapes value to `john''--` (note the two single quotes)
 - At a later point, attacker changes password (and “sets” a new password for victim john):

```
update users set password='hax' where  
database_handle("username")='john'--'
```


register.php

```
<?php
```

```
session_start();
```

```
$sql = "insert into users (username, password) values ('" .  
mysql_real_escape_string($_POST['name']) . "', '" .  
mysql_real_escape_string($_POST['password']) . "')";
```

```
mysql_query($sql);
```

```
$user_id = mysql_insert_id();
```

```
$_SESSION['uid'] = $user_id;
```

change_password.php

```
<?php
```

```
session_start();
```

```
$new_password = $_POST['password'];
```

```
$res = mysql_query("select username, password from users where  
id = '" . $_SESSION['uid'] . "'");
```

```
$row = mysql_fetch_assoc($result);
```

```
$query = "update users set password = '" .
```

```
mysql_real_escape_string($new_password) . "' where username = '"  
.$row['username'] . "' and password = '" . $row['password'] . "'";
```

```
mysql_query($query);
```

Blind SQL Injection

- A typical countermeasure is to prohibit the display of error messages: However, a web application may still be vulnerable to blind SQL injection
- Example: a news site
 - Press releases are accessed with `pressRelease.jsp?id=5`
 - A SQL query is created and sent to the database:
 - `select title, description FROM pressReleases where id=5;`
 - All error messages are filtered by the application

Blind SQL Injection

- How can we inject statements into the application and exploit it?
 - We do not receive feedback from the application so we can use a trial-and-error approach
 - First, we try to inject `pressRelease.jsp?id=5 AND 1=1`
 - The SQL query is created and sent to the database:
 - `select title, description FROM pressReleases where id=5 AND 1=1`
 - If there is a SQL injection vulnerability, the same press release should be returned
 - If input is validated, `id=5 AND 1=1` should be treated as the value

Blind SQL Injection

- When testing for vulnerability, we know $1=1$ is always true
 - However, when we inject other statements, we do not have any information
 - What we know: If the same record is returned, the statement must have been true
 - For example, we can ask server if the current user is “h4x0r”:
 - `pressRelease.jsp?id=5 AND user_name()='h4x0r'`
 - By combining subqueries and functions, we can ask more complex questions (e.g., extract the name of a database table character by character)
 - `pressRelease.jsp?id=5 AND SUBSTRING(user_name(), 1, 1) < '?'`

SQL Injection Solutions

- Developers should never allow client-supplied data to modify SQL statements
- Stored procedures
 - Isolate applications from SQL
 - All SQL statements required by the application are stored procedures on the database server
- Prepared statements
 - Statements are compiled into SQL statements before user input is added

SQL Injection Solutions:

Stored Procedures

- Original query:
 - String query = "SELECT title, description from pressReleases WHERE id= "+ request.getParameter("id");
 - Statement stat = dbConnection.createStatement();
 - ResultSet rs = stat.executeQuery(query);
- The first step to secure the code is to take the SQL statements out of the web application and **into the DB**
 - CREATE PROCEDURE getPressRelease @id integer AS SELECT title, description FROM pressReleases WHERE Id = @id

SQL Injection Solutions:

Stored Procedures

- Now, in the application, instead of string-building SQL, a stored procedure is invoked. For example, in Java:

```
CallableStatements cs = dbConnection.prepareCall(
    "{call getPressRelease(?)})");
```

```
cs.setInt(1,
    Integer.parseInt(request.getParameter("id")));
ResultSet rs = cs.executeQuery();
```


SQL Injection Solutions:

Prepared Statements

- Prepared statements allow for the clear separation of what is to be considered data and what is to be considered code
- A query is performed in a two-step process:
 - First the query is parsed and the location of the parameters identified (this is the “preparation”)
 - Then the parameters are bound to their actual values
- In some cases, prepared statements can also improve the performance of a query

SQL Injection Solutions:

Prepared Statements

```
$mysqli = new mysqli("localhost", "my_user", "my_pass", "db");  
$stmt = $mysqli->stmt_init();  
$stmt->prepare("SELECT District FROM City WHERE Name=?");  
$stmt->bind_param("s", $city);  
/* type can be "s" = string, "i" = integer ... */  
  
$stmt->execute();  
$stmt->bind_result($district);  
$stmt->fetch();  
printf("%s is in district %s\n", $city, $district);  
$stmt->close();}
```