Internet Security – COS80013 Lab - 9 Report

Internet Security – COS80013 Lab - 9 Report
Student ID: 104837257
Student Name: Arun Ragavendhar Arunachalam Palaniyappan
Lab Name: COS80013 Lab 9 – SQL injection

Date: 16 /05/2025
Tutor: Yasas Akurudda Liyanage Don

Title and Introduction

This lab was about understanding how SQL injection works and how attackers can use it to bypass logins or extract data. The setup included two virtual machines—Windows XP and RedHat Linux. From the XP machine, different SQL injection inputs were tested on a login page. The attacks worked because input checks were turned off on the server. The aim was to see how these flaws occur, what they allow, and how they can be stopped.

Methodology

The lab began by launching both the Windows XP and RedHat Linux virtual machines. On the XP browser, safelogin.php was opened and a normal login was done using "Warren" and "Eclipse". It worked as expected. After logging out, login.php was visited. This version had security checks and input sanitisations disabled.

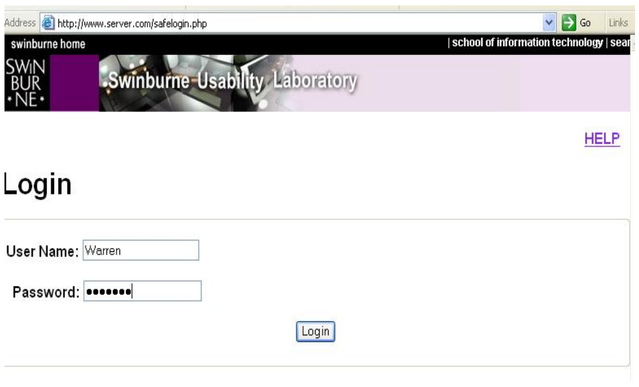
The same login failed here because the password wasn't hashed. Then, an SQL injection was tested using x' or 'x' = 'x as the password, which bypassed the login. This worked because it tricked the SQL query into always returning true.

Other common injections were tried. The string ' or 1=1;-- didn't work due to extra filtering. Column guessing was tested using x' and user = 'x, which showed an error, while x' and username = 'x did not—confirming username was valid. Table guessing was done using dot notation like users.username, which worked, while ACCOUNT.username failed.

A LIKE query using x' or username LIKE '%admin% showed that an admin-style account existed. Inputs like x' or password = '' and x' or password IS NULL-- were used to check for accounts without passwords. Advanced attack strings (like insert, update, drop) were noted but not executed, as the server wasn't set up for that level of injection.

Data Recording and Screenshots

A normal login using safelogin.php with "Warren" and "Eclipse" worked correctly.


Address http://www.server.com/safelogin.php

swinburne home | school of information technology | sear

Swinburne Usability Laboratory

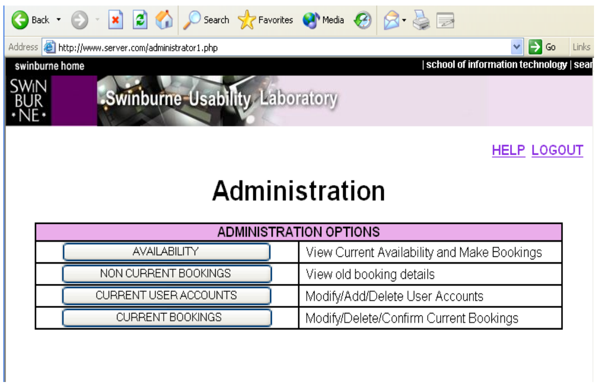
HELP

Login

User Name: Warren

Password: •••••••

Login


Address http://www.server.com/administrator1.php

swinburne home | school of information technology | sear

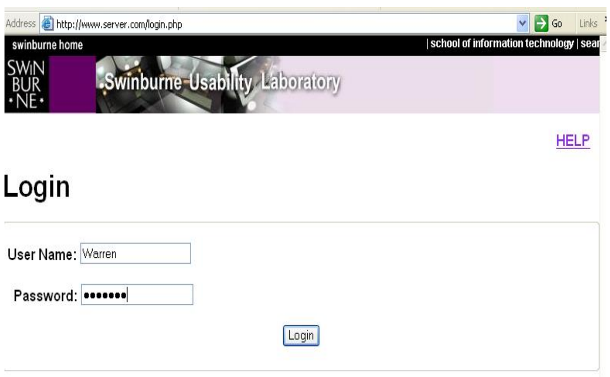
Swinburne Usability Laboratory

HELP LOGOUT

Administration

ADMINISTRATION OPTIONS	
AVAILABILITY	View Current Availability and Make Bookings
NON CURRENT BOOKINGS	View old booking details
CURRENT USER ACCOUNTS	Modify/Add/Delete User Accounts
CURRENT BOOKINGS	Modify/Delete/Confirm Current Bookings

The same login failed in login.php due to hashing differences.


Address http://www.server.com/login.php

swinburne home | school of information technology | sear

Swinburne Usability Laboratory

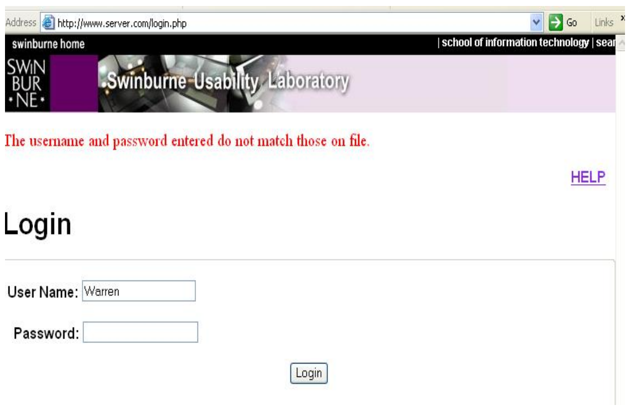
HELP

Login

User Name: Warren

Password: •••••••

Login


Address http://www.server.com/login.php

swinburne home | school of information technology | sear

Swinburne Usability Laboratory

The username and password entered do not match those on file.

HELP

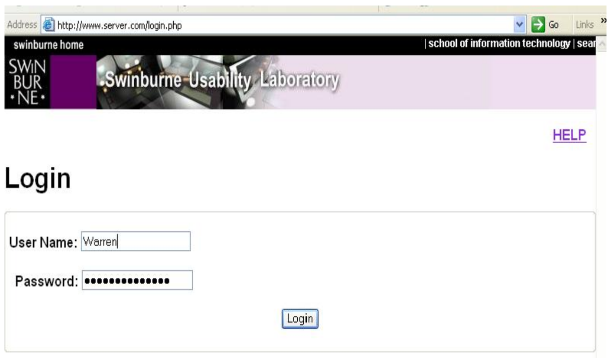
Login

User Name: Warren

Password:

Login

The input x' or 'x' = 'x successfully bypassed the login in login.php.


Address http://www.server.com/login.php

swinburne home | school of information technology | sear

Swinburne Usability Laboratory

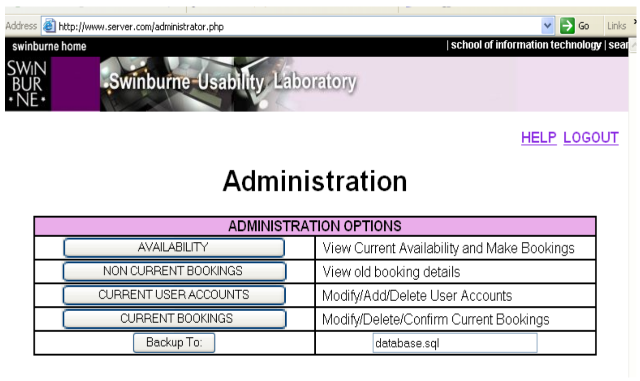
HELP

Login

User Name: Warren

Password: ••••••••••••••

Login


Address http://www.server.com/administrator.php

swinburne home | school of information technology | sear

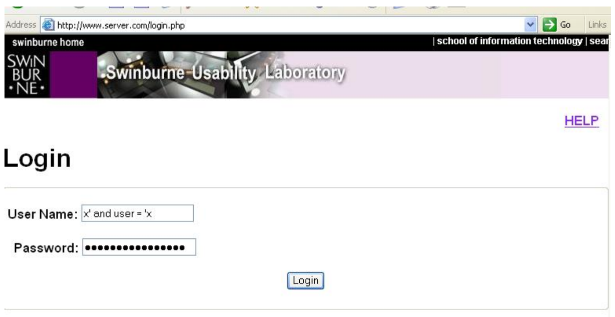
Swinburne Usability Laboratory

HELP LOGOUT

Administration

ADMINISTRATION OPTIONS	
AVAILABILITY	View Current Availability and Make Bookings
NON CURRENT BOOKINGS	View old booking details
CURRENT USER ACCOUNTS	Modify/Add/Delete User Accounts
CURRENT BOOKINGS	Modify/Delete/Confirm Current Bookings
Backup To:	database.sql

Trying x' and user = 'x showed a database error.


Address http://www.server.com/login.php

swinburne home | school of information technology | sear

Swinburne Usability Laboratory

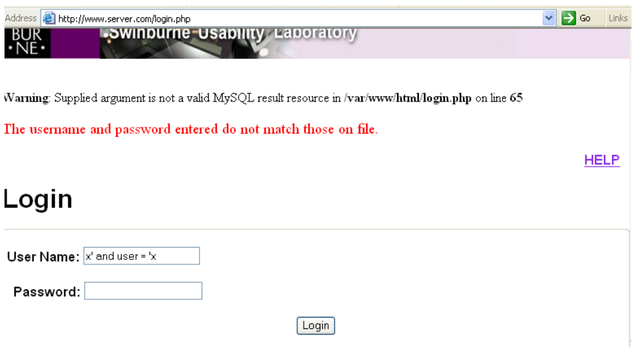
HELP

Login

User Name: x' and user = 'x

Password: ••••••••••••••••

Login


Address http://www.server.com/login.php

Swinburne Usability Laboratory

Warning: Supplied argument is not a valid MySQL result resource in /var/www/html/login.php on line 65

The username and password entered do not match those on file.

HELP

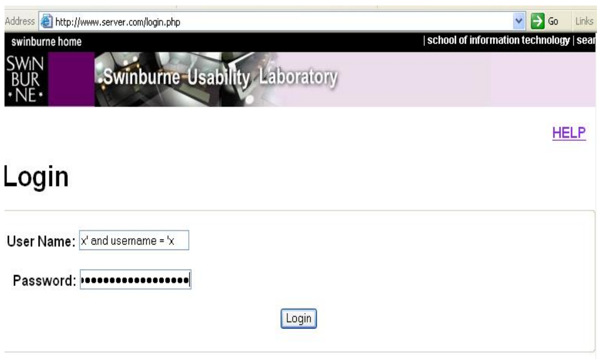
Login

User Name: x' and user = 'x

Password:

Login

Trying x' and username = 'x did not show a database error, confirming that the column existed.


Address http://www.server.com/login.php

swinburne home | school of information technology | sear

Swinburne Usability Laboratory

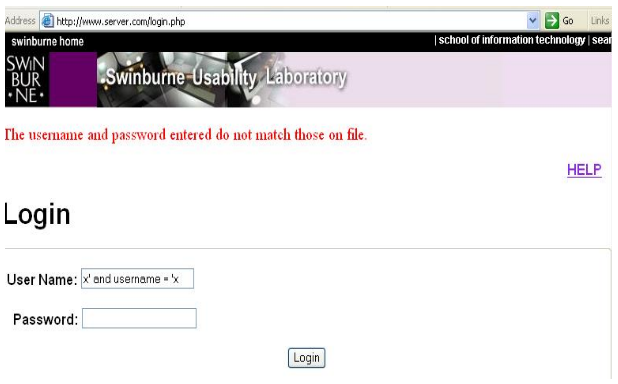
HELP

Login

User Name: x' and username = 'x

Password: ••••••••••••••••••

Login


Address http://www.server.com/login.php

swinburne home | school of information technology | sear

Swinburne Usability Laboratory

The username and password entered do not match those on file.

HELP

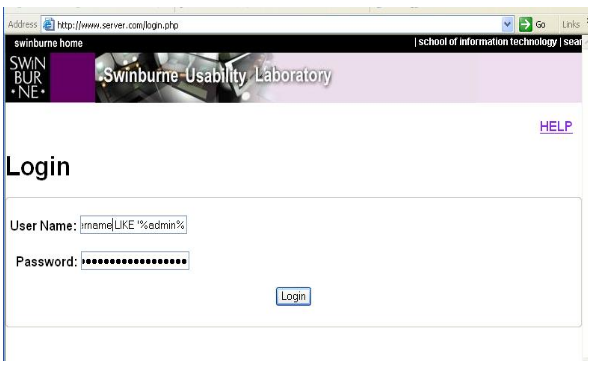
Login

User Name: x' and username = 'x

Password:

Login

Using x' or username LIKE '%admin% revealed that an admin-like user existed.


Address http://www.server.com/login.php

swinburne home | school of information technology | sear

Swinburne Usability Laboratory

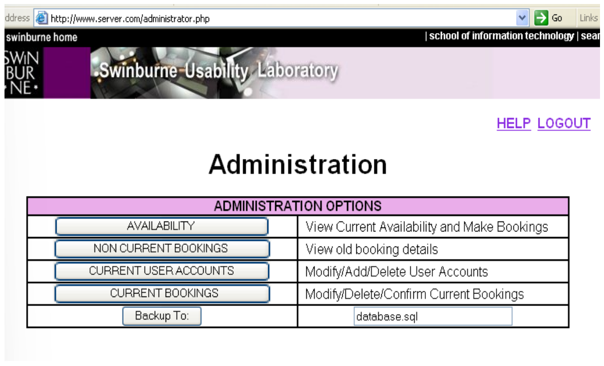
HELP

Login

User Name: rname LIKE '%admin%

Password: ••••••••••••••••••

Login


ddress http://www.server.com/administrator.php

swinburne home | school of information technology | sear

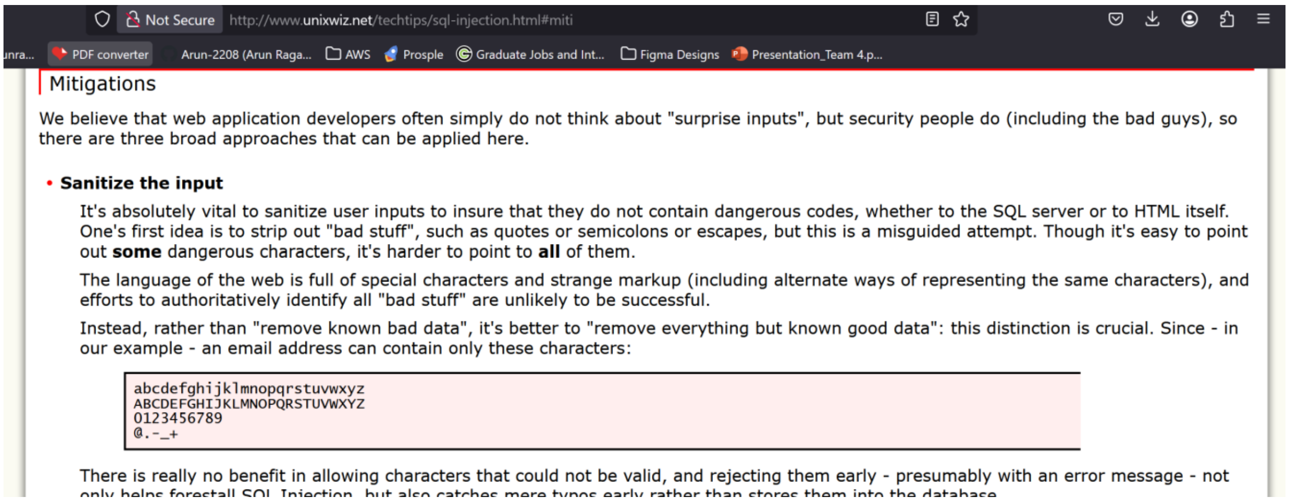
Swinburne Usability Laboratory

HELP LOGOUT

Administration

ADMINISTRATION OPTIONS	
AVAILABILITY	View Current Availability and Make Bookings
NON CURRENT BOOKINGS	View old booking details
CURRENT USER ACCOUNTS	Modify/Add/Delete User Accounts
CURRENT BOOKINGS	Modify/Delete/Confirm Current Bookings
Backup To:	database.sql

Studied about the 7 techniques to prevent SQL injection attacks from http://www.unixwiz.net/techtips/sql-injection.html


Not Secure http://www.unixwiz.net/techtips/sql-injection.html#miti

Mitigations

We believe that web application developers often simply do not think about "surprise inputs", but security people do (including the bad guys), so there are three broad approaches that can be applied here.

- **Sanitize the input**

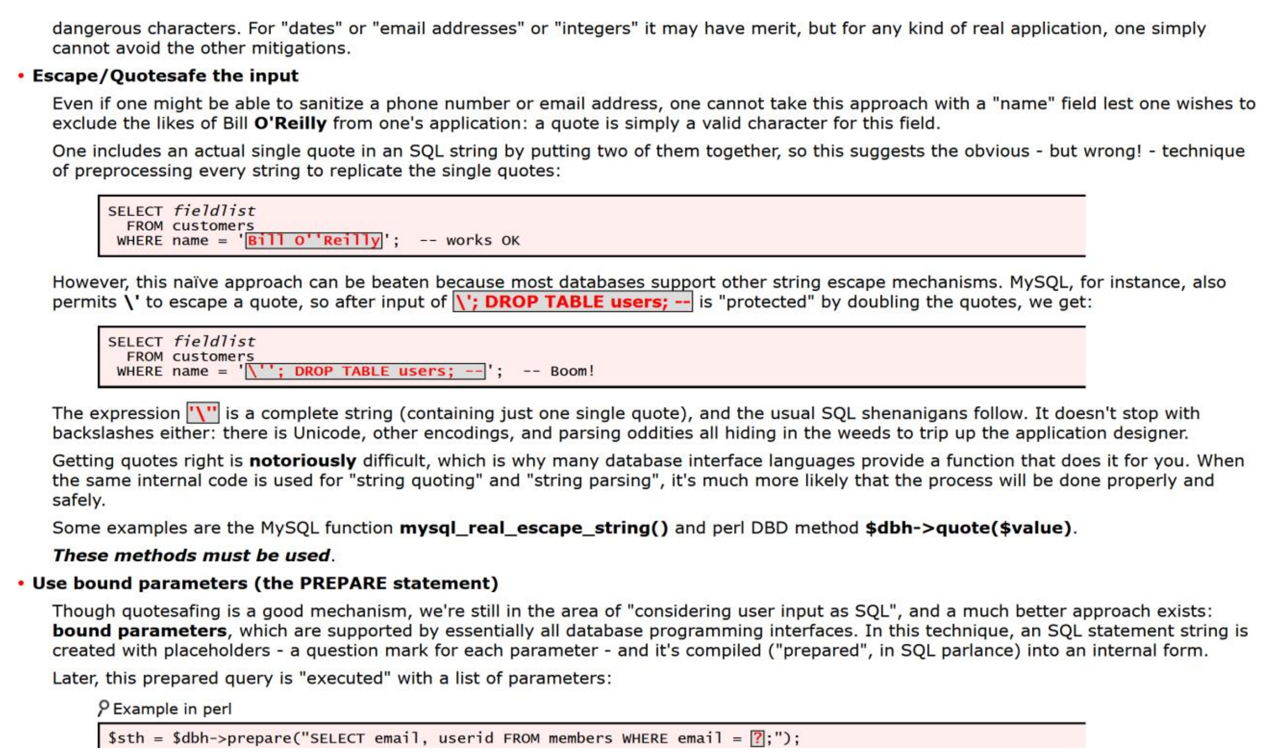
  It's absolutely vital to sanitize user inputs to insure that they do not contain dangerous codes, whether to the SQL server or to HTML itself. One's first idea is to strip out "bad stuff", such as quotes or semicolons or escapes, but this is a misguided attempt. Though it's easy to point out **some** dangerous characters, it's harder to point to **all** of them.

  The language of the web is full of special characters and strange markup (including alternate ways of representing the same characters), and efforts to authoritatively identify all "bad stuff" are unlikely to be successful.

  Instead, rather than "remove known bad data", it's better to "remove everything but known good data": this distinction is crucial. Since - in our example - an email address can contain only these characters:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
@.-_+
```

  There is really no benefit in allowing characters that could not be valid, and rejecting them early - presumably with an error message - not only helps forestall SQL Injection, but also catches mere typos early rather than stores them into the database.


dangerous characters. For "dates" or "email addresses" or "integers" it may have merit, but for any kind of real application, one simply cannot avoid the other mitigations.

- **Escape/Quotesafe the input**

  Even if one might be able to sanitize a phone number or email address, one cannot take this approach with a "name" field lest one wishes to exclude the likes of Bill **O'Reilly** from one's application: a quote is simply a valid character for this field.

  One includes an actual single quote in an SQL string by putting two of them together, so this suggests the obvious - but wrong! - technique of preprocessing every string to replicate the single quotes:

```
SELECT fieldlist
  FROM customers
 WHERE name = 'Bill O''Reilly';  -- works OK
```

  However, this naïve approach can be beaten because most databases support other string escape mechanisms. MySQL, for instance, also permits **\'** to escape a quote, so after input of **\'; DROP TABLE users; --** is "protected" by doubling the quotes, we get:

```
SELECT fieldlist
  FROM customers
 WHERE name = '\''; DROP TABLE users; --';  -- Boom!
```

  The expression **'\''** is a complete string (containing just one single quote), and the usual SQL shenanigans follow. It doesn't stop with backslashes either: there is Unicode, other encodings, and parsing oddities all hiding in the weeds to trip up the application designer.

  Getting quotes right is **notoriously** difficult, which is why many database interface languages provide a function that does it for you. When the same internal code is used for "string quoting" and "string parsing", it's much more likely that the process will be done properly and safely.

  Some examples are the MySQL function **mysql_real_escape_string()** and perl DBD method **$dbh->quote($value)**.

  These methods must be used.

- **Use bound parameters (the PREPARE statement)**

  Though quotesafing is a good mechanism, we're still in the area of "considering user input as SQL", and a much better approach exists: **bound parameters**, which are supported by essentially all database programming interfaces. In this technique, an SQL statement string is created with placeholders - a question mark for each parameter - and it's compiled ("prepared", in SQL parlance) into an internal form.

  Later, this prepared query is "executed" with a list of parameters:

  Example in perl

```
$sth = $dbh->prepare("SELECT email, userid FROM members WHERE email = ?;");
```

Discussion and Application of Learnings

Learning 1
The string x' or 'x' = 'x was able to log in without knowing the password. It worked because the SQL query was changed to always return true.
Real-World Application
If a website does not properly clean input fields, attackers can bypass logins and gain access as any user.

Learning 2
Using made-up column names like user caused an error, but using real ones like username did not. This showed that attackers can guess a system’s database structure.
Real-World Application
By watching how the system responds to errors, attackers can slowly map out database columns and tables.

Learning 3
The input using LIKE helped find usernames that contain certain words. This makes it possible to identify accounts like “admin” without needing a full username.
Real-World Application
This method can help attackers find target accounts or test for weak usernames.

Learning 4
Testing for empty passwords showed that accounts with no password could be exploited.
Real-World Application
Some systems don’t force password creation, so attackers can find and take over these accounts.

Learning 5
Some SQL commands like insert, update, or drop were not run here, but they show how much damage is possible if input is not filtered.
Real-World Application
In real attacks, these kinds of commands can change data, create fake users, or even delete entire tables.

Limitations

The lab used a purposely vulnerable setup in a safe environment. Real systems often block such inputs using proper sanitisation. Some injections didn’t work due to extra filters still being active. Destructive commands like drop table were noted but not executed. Still, the lab clearly showed how SQL injection works and why input handling matters.