# Database Security

# SQL Injection

- Achieved by changing the structure of the database request
  - Redesigning the SQL statement
  - Uses single quotes to add additional structure to query
- MySQL
  - Has command line interface that works with MariaDB as well

# Create Database in MySQL

- Log in to Debian as root:
- **mysql –u root**
- **create database mydb;**
- **grant all privileges on mydb.* to ‘mydbadmin'@'%' identified by 'password' with grant option;**

# Create Table

- Connect to your database using SQLManager and create the following;

```
create table users
(
   userId int KEY AUTO_INCREMENT,
   userName varchar(50) not null,
   userPass varchar(20) not null
);
```

- Create two records, one for John, one for admin.

# Sample SQL Injections

- `select count(*) from users where userName='john' and userPass='doe';`
- `select count(*) from users where userName='john' and userPass='' or 1=1 #' ;`
  - Inserts <u>`' or 1=1 #`</u> in place of password
- `select count(*) from users where userName='' or 1=1 #' and userPass='' ;`
- `select * from users where userName='john' and userPass='' or 1=1 #' ;`

# PHP and HTML Objects

- For our purposes, we will create a HTML form and PHP to access a list of users, and log them in if the username and password match
  - sql_injection_form.html and login.php in Learn
- Files should be in the document root folder for XAMPP, and accessed with http://localhost/
  - Requires that Apache and MySQL be running

# SQL Injection in PHP

- Form structure:

```
<form name="frmLogin" action="login.php"
method="get">
<p>Username: <input type="text"
name="userName">
<p>Password: <input type="text"
name="password">
<p><input type="submit">
</form>
```

# SQL Injection in PHP (cont)

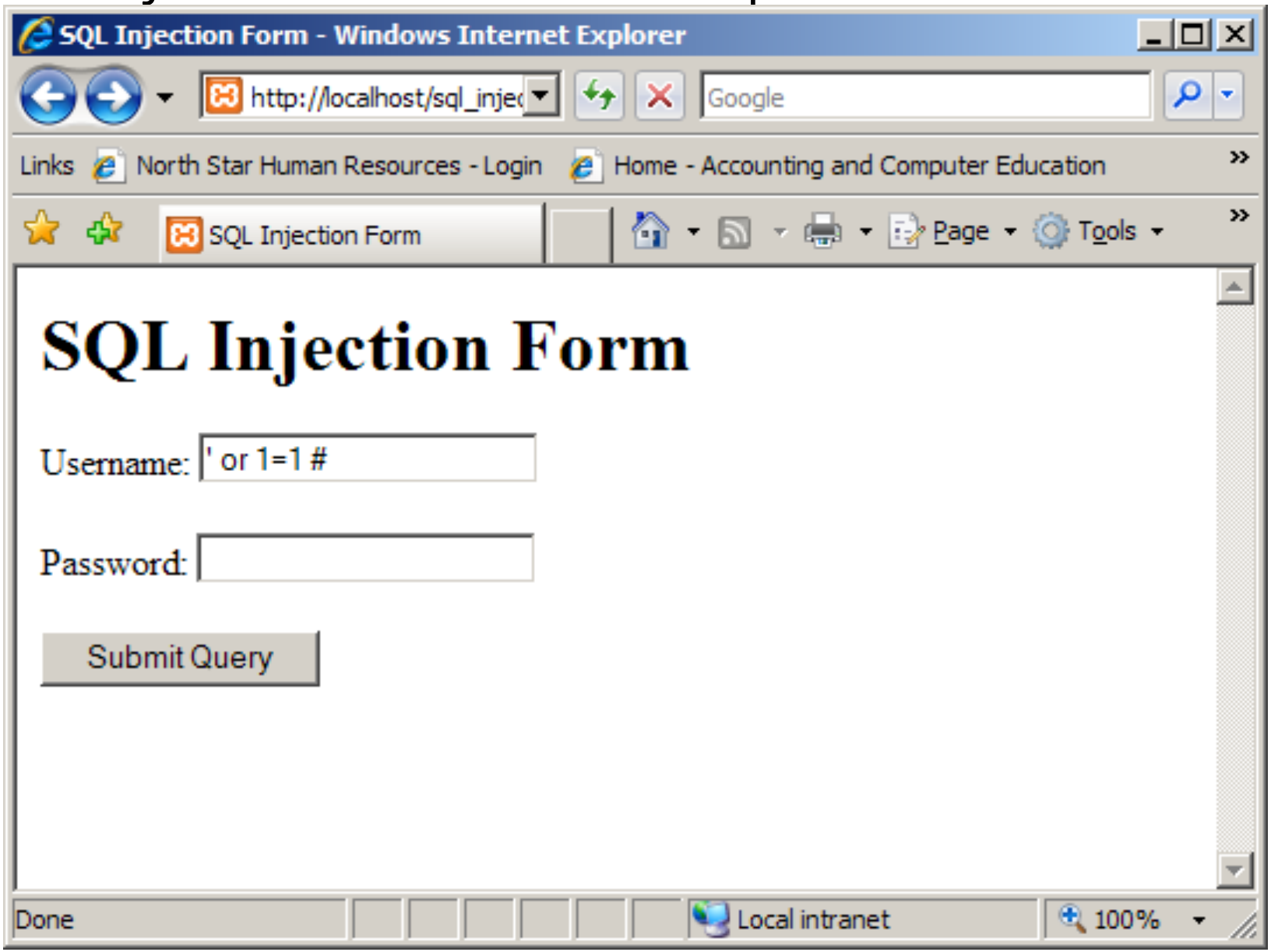
```
<html><body><h1>SQL Injection Results</h1>
<div style="border: 1px solid #000000;
 width :230px; margin-top:
 50px;margin-left: 70px;
 padding:20px 20px 20px 20px ;
 background-color: #F5F5FF;">
<?php
 $host = "localhost";
 $user = "mydbadmin";
 $password = "password";
 $database = "mydb";
$mysqli = new mysqli($host, $user, $password, $database);
/* check connection */
if ($mysqli->connect_errno) {
    printf("Connect failed: %s\n", $mysqli->connect_error);
    exit();
}
 if (isset($_GET['userName'])) {
     $userName=$_GET["userName"];
     $password=$_GET["password"];
```

# SQL Injection in PHP (cont)

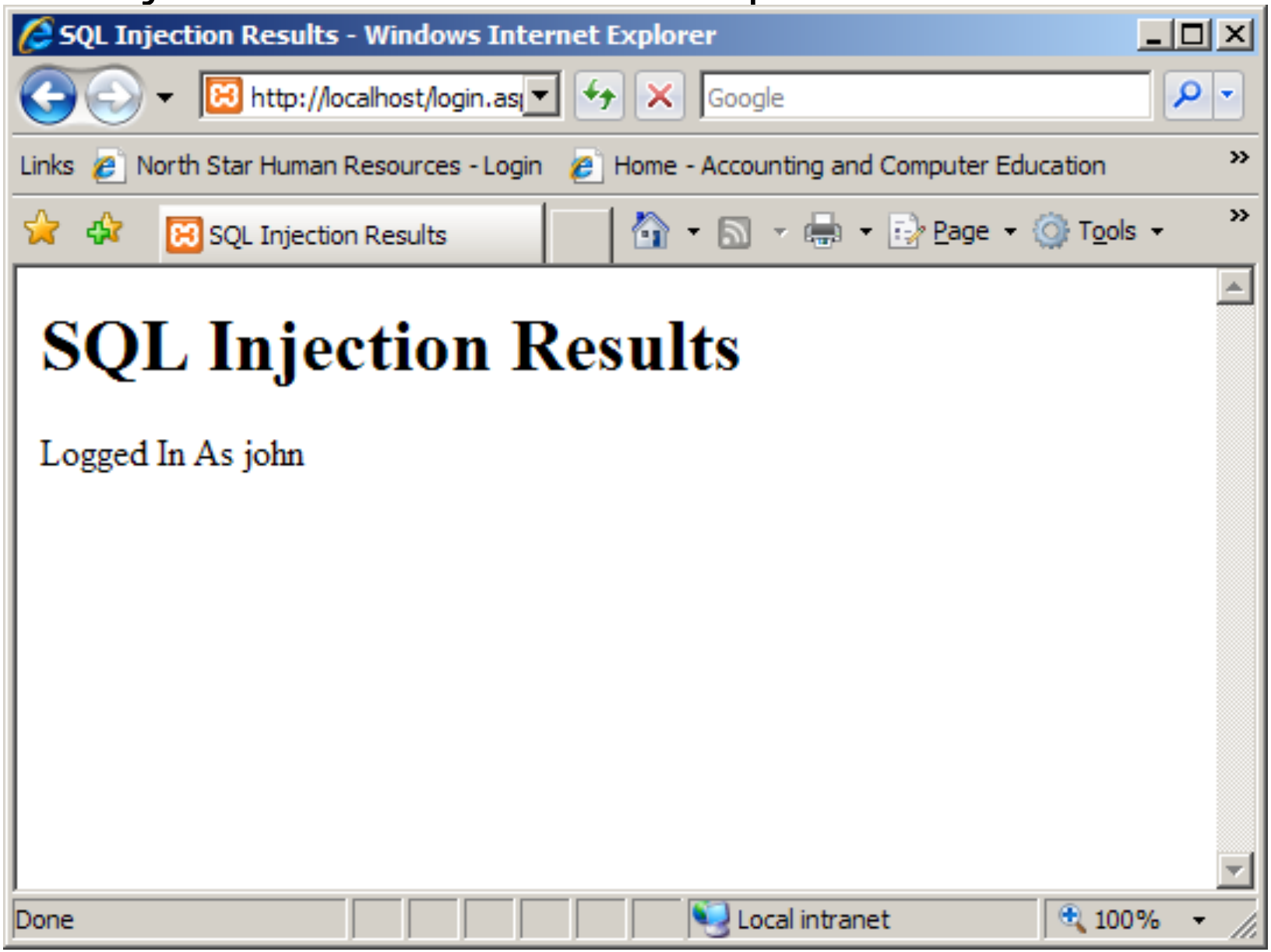
```
    $sql="select userName from users \nwhere
 userName='" . $userName . "' \n and userPass='" .
 $password . "'";
 $result = $mysqli->query($sql);
 $sql_num=0;
 while ($row = $result->fetch_assoc())
 {
    echo "<p>Logged in as ".$row["userName"];
    ++$sql_num;
 }
 if ($sql_num==0)
 {
    echo "<p>Bad credentials";
 }
    mysqli_close ( $mysqli );

 }
?>
</div></body></html>
```

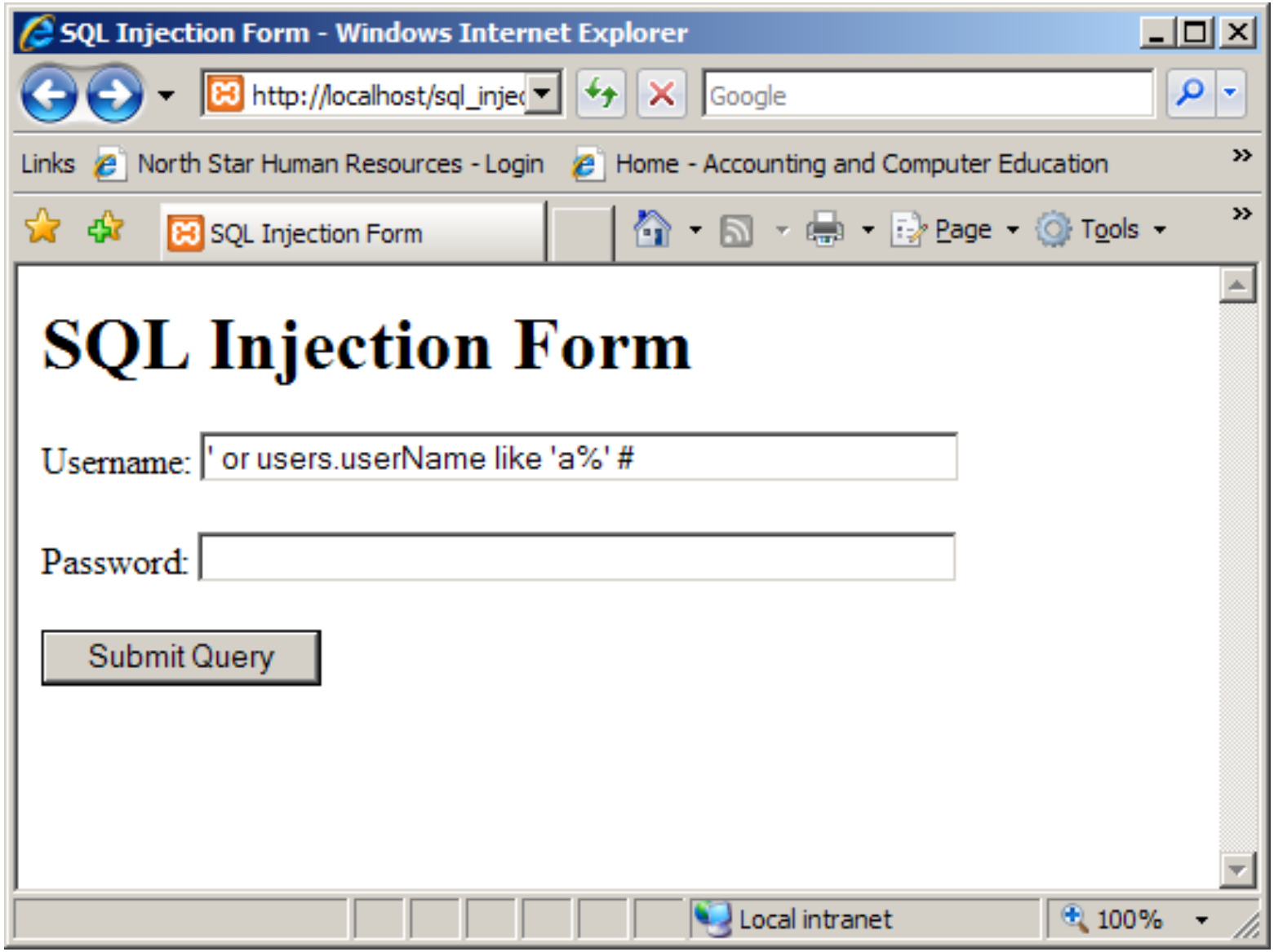
# SQL Injection in PHP Example 1



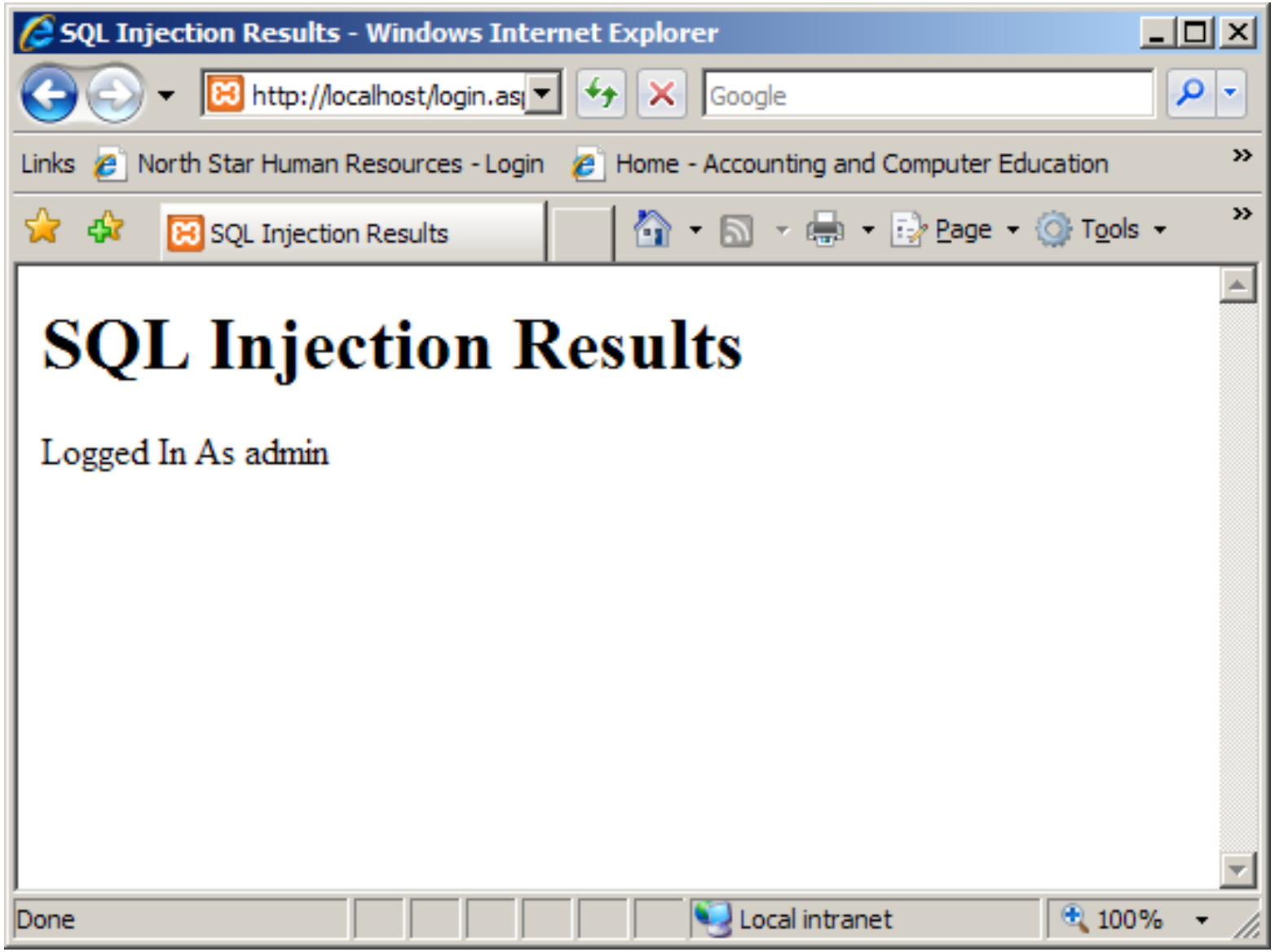
# SQL Injection in PHP Example 1



# SQL Injection in PHP Example 2



# SQL Injection in PHP Example 2



# SQL Results of Injections

- `select userName from users where userName='john' and userPass='' or 1=1 #'`

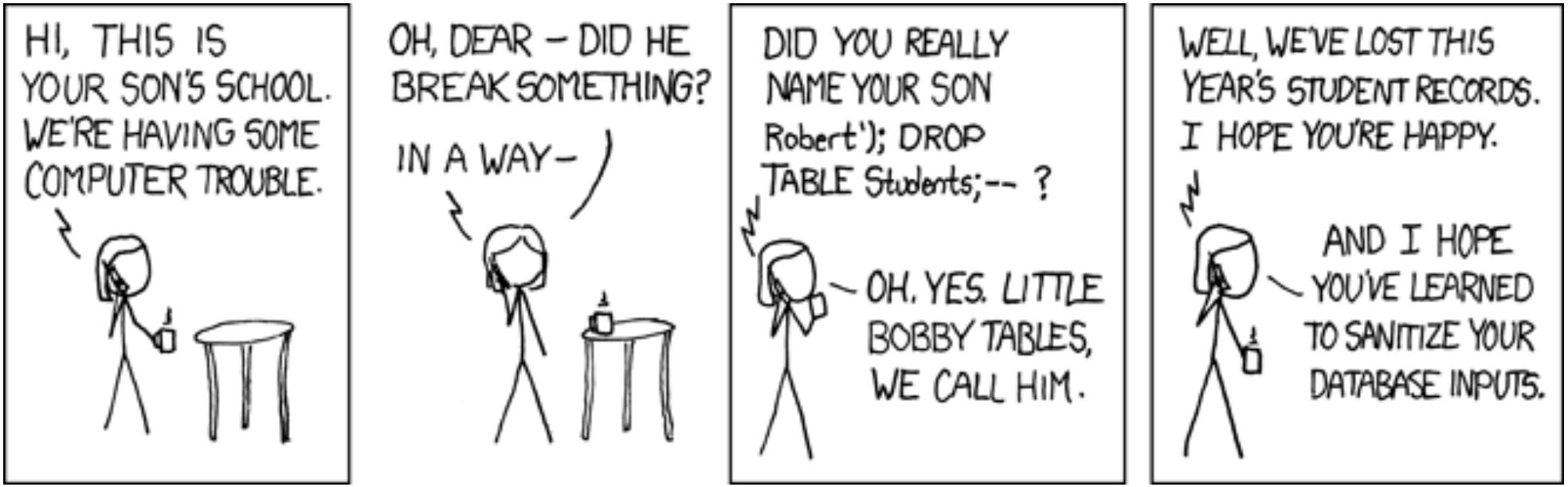
- `select userName from users where userName='' or users.userName like 'a%' #' and userPass=''`

- In most cases, the comment will be the # or the -- character(s). With MariaDB/MySQL, the # works better.

# Other Database Examples

- May be out of date, but serve as a warning
- In MS SQL Server, the following is possible
  - '; exec master..xp_cmdshell 'iisreset'; --
  - '; exec master..xp_cmdshell 'format c: /q /yes '; drop database myDB; --
- MS Access has similar functionality

# Now this should make sense…



# Roles and Views

- One way to protect on-line data is to setup users into the database that don’t have wide open access.
  - Different database systems have different methods of granting access to different items, but most are controlled with the grant command
  - Setup “webuser” accounts to simply query tables, or access predefined queries/views, and don’t give them the ability to create new users/tables/objects

# MySQL Example

- Database Level:
```
GRANT SELECT, INSERT ON mydb.* TO
'someuser'@'somehost';
```
- Table Level:
```
GRANT SELECT, INSERT ON mydb.mytbl TO
'someuser'@'somehost';
```
- Column Level:
```
GRANT SELECT (col1), INSERT (col1,col2) ON
mydb.mytbl TO 'someuser'@'somehost';
```

- http://dev.mysql.com/doc/refman/5.0/en/grant.html

# Filter out threats

- One of the most common forms of SQL injection involves inserting a single quote to terminate an expression. A simple replace function can be used to change one quote into two:

```
function stripQuotes(strWords)
   stripQuotes = replace(strWords, "'",
           "''")
end function
```

# Filter out threats (cont)

- One can create a simple function that removes potentially dangerous code from input text

```
function killChars(strWords)
dim badChars
dim newChars
badChars = array [4]("select", "drop", ";", “#", "insert",  "delete", "xp_")
newChars = strWords
for i = 0 to uBound(badChars)
    newChars = replace(newChars, badChars(i), "")
next
killChars = newChars
end function
```

# Limit input sizes

- If you limit inputs for text fields to the size expected, you can add a layer of protection
  - Will quantity sizes ever be more than 2 or 3 digits?
  - Will usernames and/or passwords be more than 10 characters?
- Set your form method to Post and not Get
  - Limits the ability to craft injections in the URL
- Validate data types – is a number input a number?

# Use a Database Abstraction Library

- One of the easiest and probably best ways to get around the threats of SQL injection is to use some kind of library that analyzes form inputs for potential threats.
- Libraries typically come with frameworks (CakePHP, Struts, SQLpp for C#, ASP.NET frameworks)

# PDO for PHP

- PDO is probably one of the more popular DB abstractions
  - Included with PHP
  - Gives you the ability to use parameters and binding

# Line Breaks

- Another great solution is to use line breaks in your SQL code
- Interrupts the comment to end of line component of SQL injections