Relational Databases and Web Integration

Dr. Carl Pulley
c.j.pulley@hud.ac.uk

SQL Injection

- Poor sanitation of user inputs can often allow an attacker to inject SQL statements into a web application leading to:
  - reading sensitive database data
  - modifying database
  - execute admin operations on database
  - read local files
  - execute OS level code

Basic SQL Injection

```
<form action="LoginAuthBypass.php" method="post">
  Username: <input type="text" name="username" /> <BR/>
  Password: <input type="password" name="password" /> <BR/>
  <input type="submit" value="Login" />
</form>
```

```
<?php
..
$username = $_POST['username'];
$password = $_POST['password'];
..
$loginauthsql = "SELECT * FROM USERS WHERE USER_NAME = '" . $username . "' AND
PASSWORD = '" . $password . "'";
..
?>
```

`"' OR 'z'='z"` `"admin"`

```
$loginauthsql = "SELECT * FROM USERS WHERE USER_NAME = 'admin' AND PASSWORD = '' OR 'z'='z'"
$loginauthsql = "SELECT * FROM USERS WHERE TRUE"
```


Alternative injections strings:

$username = "' OR 1=1 --"

SQL Injection Types

- Inband
  - data extracted using the injection channel (eg. data embedded in web page)
- Out-of-band
  - data extracted using another channel (eg. email)
- Inferential
  - DB behaviour allows one to infer information

PHP Magic Quotes

- Flawed attempt at trying to stop SQL injection
  - escape ‘, “, \ and `null` characters
  - hex encoding injection string gets around the use of this PHP directive!
- Not all strings need escaping
  - developer can spend a lot of time undoing the damage this directive causes!

Tuesday, 19 January 2010

PHP magic quotes is depreciated in PHP 5 and removed in PHP 6.

PHP/MySQL Interaction

- Traditionally interact with MySQL via the PHP library functions:
  - `mysql_*`
  - `mysqli_*`
- These functions offer **no** protection against SQL injection attacks!
- These functions are very DB specific
  - hard to change the underlying DB

PHP Data Objects

- PDO provides a database abstraction layer
  - easy to change underlying DB with minimal impact on your web application

```
$db = new PDO(‘mysql:host=localhost; dbname=u1234567’, $user, $pw);
..
$db = null;
```

Tuesday, 19 January 2010

PHP variables start with a $ symbol and all PHP statements should end with a semi-colan.
PDO here is a PHP class and new PDO(..) creates a new object instance of that class. Setting a variable to null allows the object to be garbaged collected (ie. the DB connection to be closed).
By altering the PDO connection string (ie. ‘mysql:...’) we can easily change the underlying DB with minimal impact on our web application.
Generally speaking, it is a very bad idea to allow any database to accept remote network connections - it increases your servers attack surface!

DB Connection Credentials

- Place in a file that is included within your main PHP application code.
  - do **not** hard code these into web accessible code!
  - credential file should **not** be present within Apache web directories!
  - needs to be readable and accessible by Apache

Tuesday, 19 January 2010

On helios, connection credentials should be placed within /spare/pwds - this directory can be accessed by Apache, but it is not web accessible!

Prepared Statements

- The **only** safe way to avoid SQL injection is to use prepared statements

```
$stmt = $db->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");
$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);

$name = $_GET[‘name’];
$value = $_GET[‘value’];
$stmt->execute();
```

```
$stmt = $db->prepare("SELECT * FROM REGISTRY where name = ?");
if ($stmt->execute(array($_GET['name']))) {
  while ($row = $stmt->fetch()) {
    print_r($row);
  }
}
```

Tuesday, 19 January 2010

-> is used to access a component (eg. attribute or method) of an object instance in PHP.
$_GET is a PHP array that holds the variables set by a HTTP GET request.
$_POST is a PHP array that holds the variables set by a HTTP POST request.
$_REQUEST is a PHP array that holds the variables set by a HTTP GET or POST request.
array(..) generates a PHP array - in this case, with one element which is used to bind to the ? parameter (ie. position 1 in our prepared statement). Multiple ?’s in our prepared statement would require an array with multiple arguments.

Prepared Statements

```
    $stmt = $db->prepare('SELECT name, colour, calories
        FROM fruit
        WHERE calories < :calories AND colour = :colour');
    $stmt->execute(array(':calories' => $calories, ':colour' => $colour));
```

```
$count = $db->exec("DELETE FROM fruit WHERE colour = 'red'");
print("Deleted $count rows.\n"); /* Return number of rows that were deleted */
```

```
function getFruit($conn) {
    $sql = 'SELECT name, color, calories FROM fruit ORDER BY name';
    foreach ($conn->query($sql) as $row) {
        print $row['name'] . "\t";
        print $row['color'] . "\t";
        print $row['calories'] . "\n";
    }
}
```