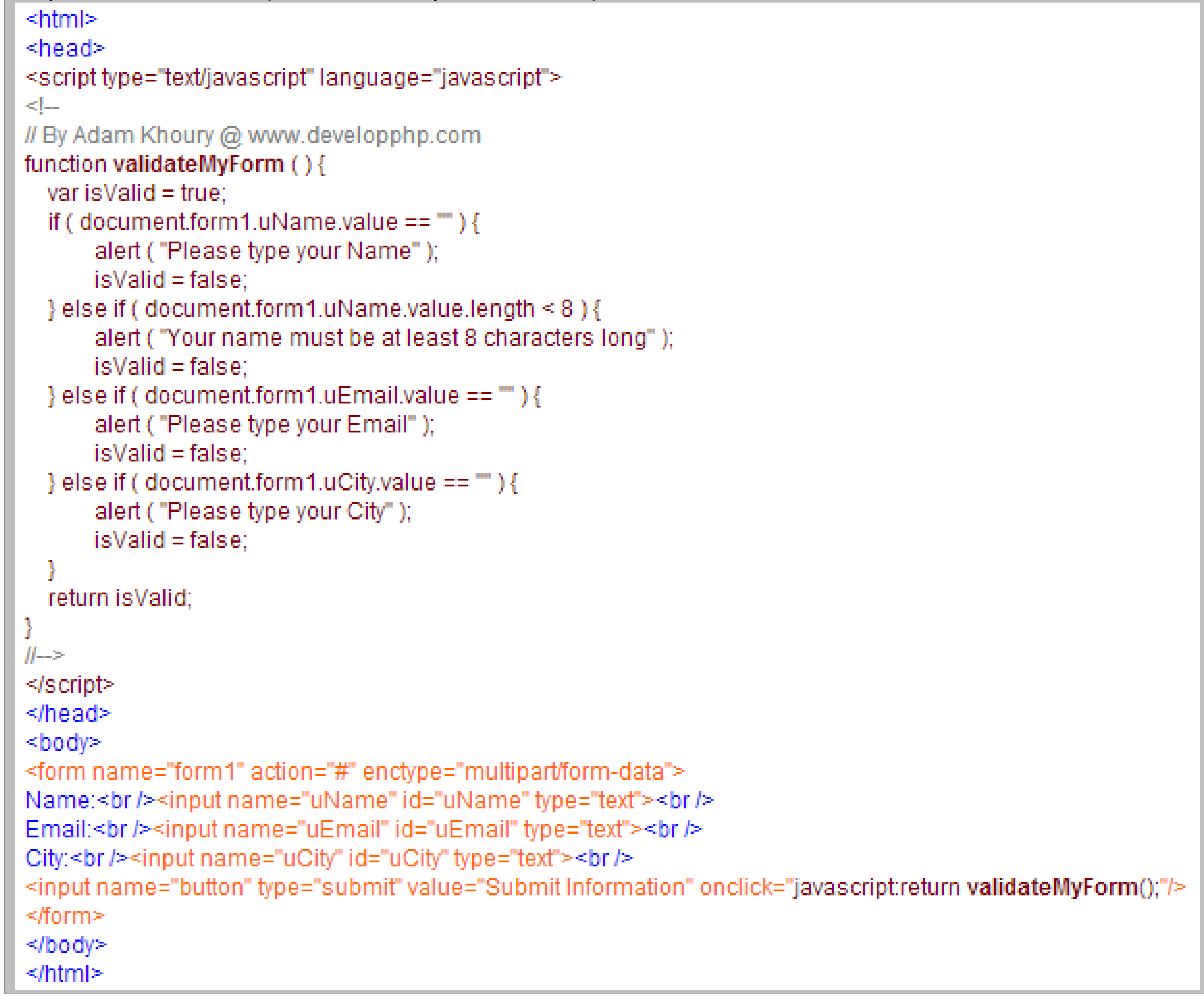
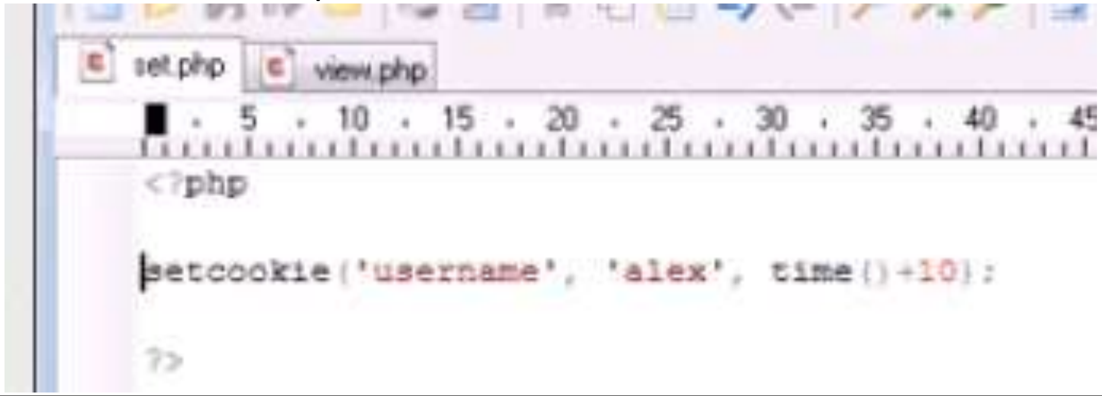
Simple PHP form validation (from Adam Khoury Tutorial website)



```
<html>
<head>
<script type="text/javascript" language="javascript">
<!--
// By Adam Khoury @ www.developphp.com
function validateMyForm ( ) {
   var isValid = true;
   if ( document.form1.uName.value == "" ) {
         alert ( "Please type your Name" );
         isValid = false;
   } else if ( document.form1.uName.value.length < 8 ) {
         alert ( "Your name must be at least 8 characters long" );
         isValid = false;
   } else if ( document.form1.uEmail.value == "" ) {
         alert ( "Please type your Email" );
         isValid = false;
   } else if ( document.form1.uCity.value == "" ) {
         alert ( "Please type your City" );
         isValid = false;
   }
   return isValid;
}
//-->
</script>
</head>
<body>
<form name="form1" action="#" enctype="multipart/form-data">
Name:<br /><input name="uName" id="uName" type="text"><br />
Email:<br /><input name="uEmail" id="uEmail" type="text"><br />
City:<br /><input name="uCity" id="uCity" type="text"><br />
<input name="button" type="submit" value="Submit Information" onclick="javascript:return validateMyForm();"/>
</form>
</body>
</html>
```

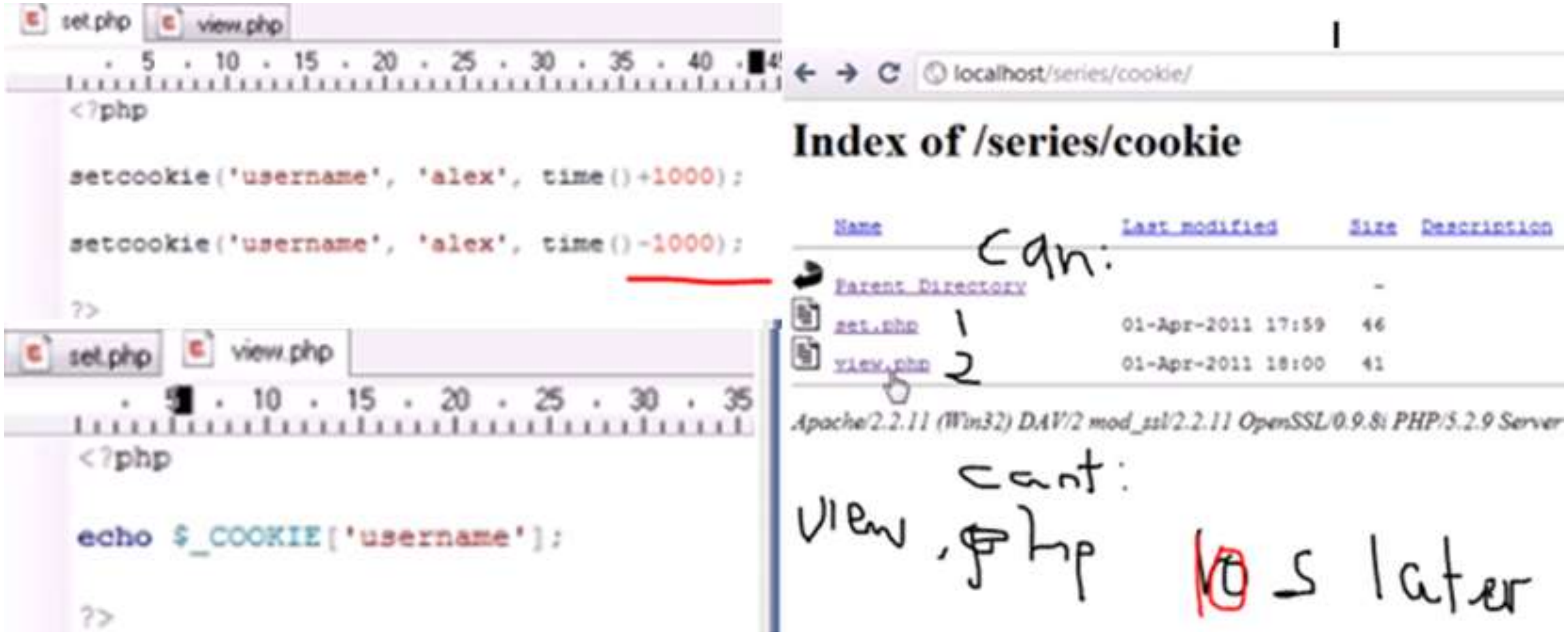
Create Cookies with PHP | TheNewBoston Tutorials



set.php | view.php

```php
<?php

setcookie('username', 'alex', time()+10);

?>
```

Delete Cookie with PHP:



set.php | view.php

```php
<?php

setcookie('username', 'alex', time()+1000);

setcookie('username', 'alex', time()-1000);

?>
```

set.php | view.php

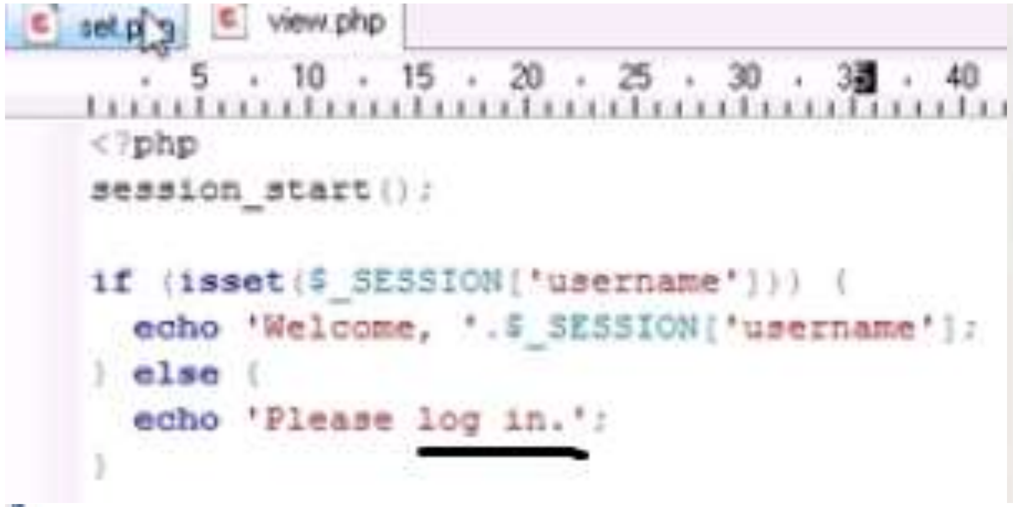
```php
<?php

echo $_COOKIE['username'];

?>
```

localhost/series/cookie/

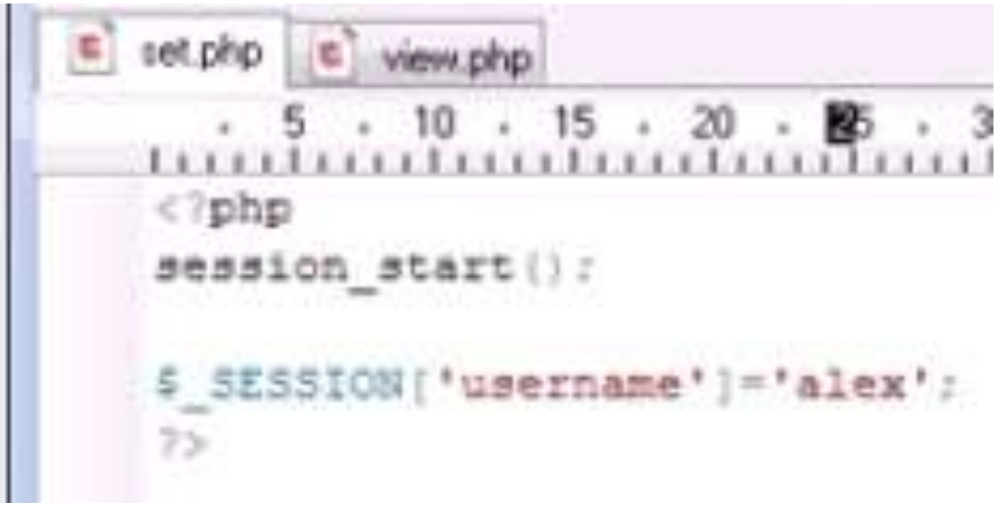
Index of /series/cookie

```
Name                  Last modified        Size  Description
Parent Directory                            -
set.php    1          01-Apr-2011 17:59    46
view.php   2          01-Apr-2011 18:00    41

Apache/2.2.11 (Win32) DAV/2 mod_ssl/2.2.11 OpenSSL/0.9.8i PHP/5.2.9 Server
```

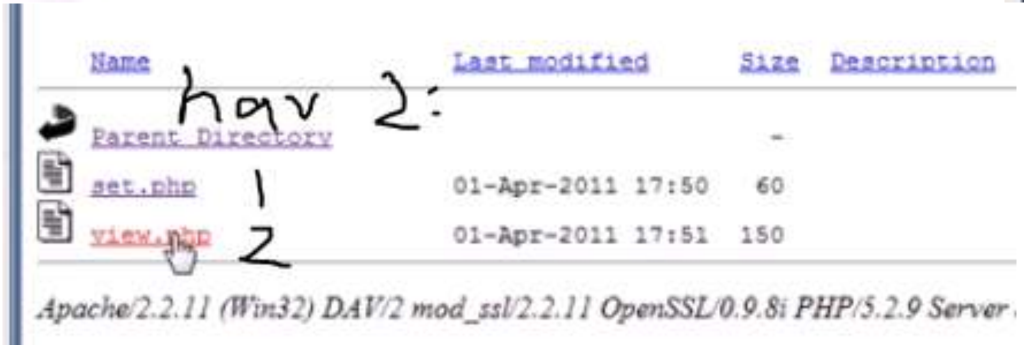
Set Sessions:



```
<?php
session_start();

if (isset($_SESSION['username'])) {
  echo 'Welcome, '.$_SESSION['username'];
} else {
  echo 'Please log in.';
}
```



```
<?php
session_start();

$_SESSION['username']='alex';
?>
```

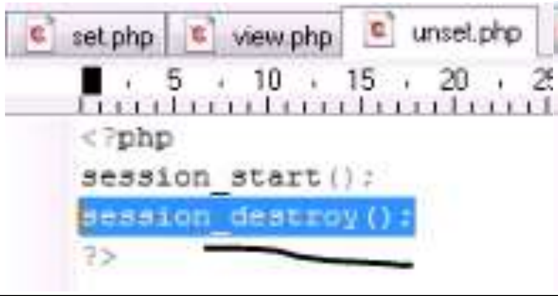


hqv 2:

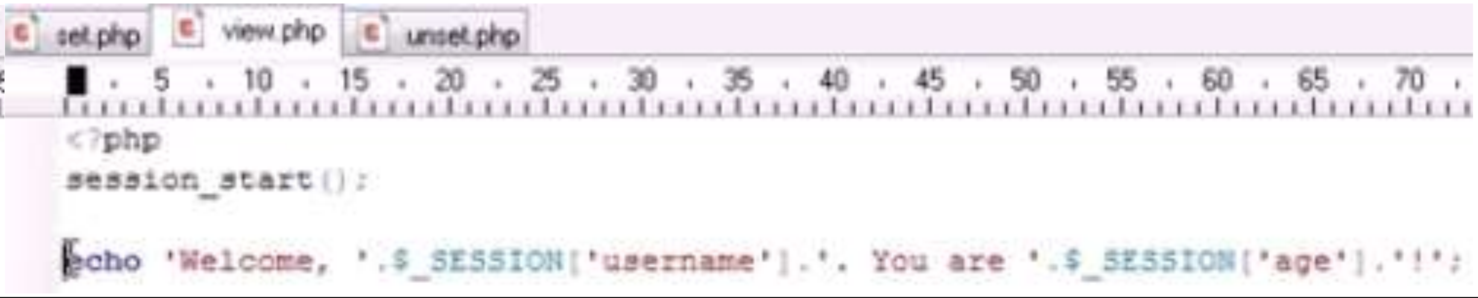
```
Name                 Last modified      Size  Description
Parent Directory                          -
set.php    1         01-Apr-2011 17:50   60
view.php   2         01-Apr-2011 17:51  150
```

Apache/2.2.11 (Win32) DAV/2 mod_ssl/2.2.11 OpenSSL/0.9.8i PHP/5.2.9 Server

Unset Sessions:

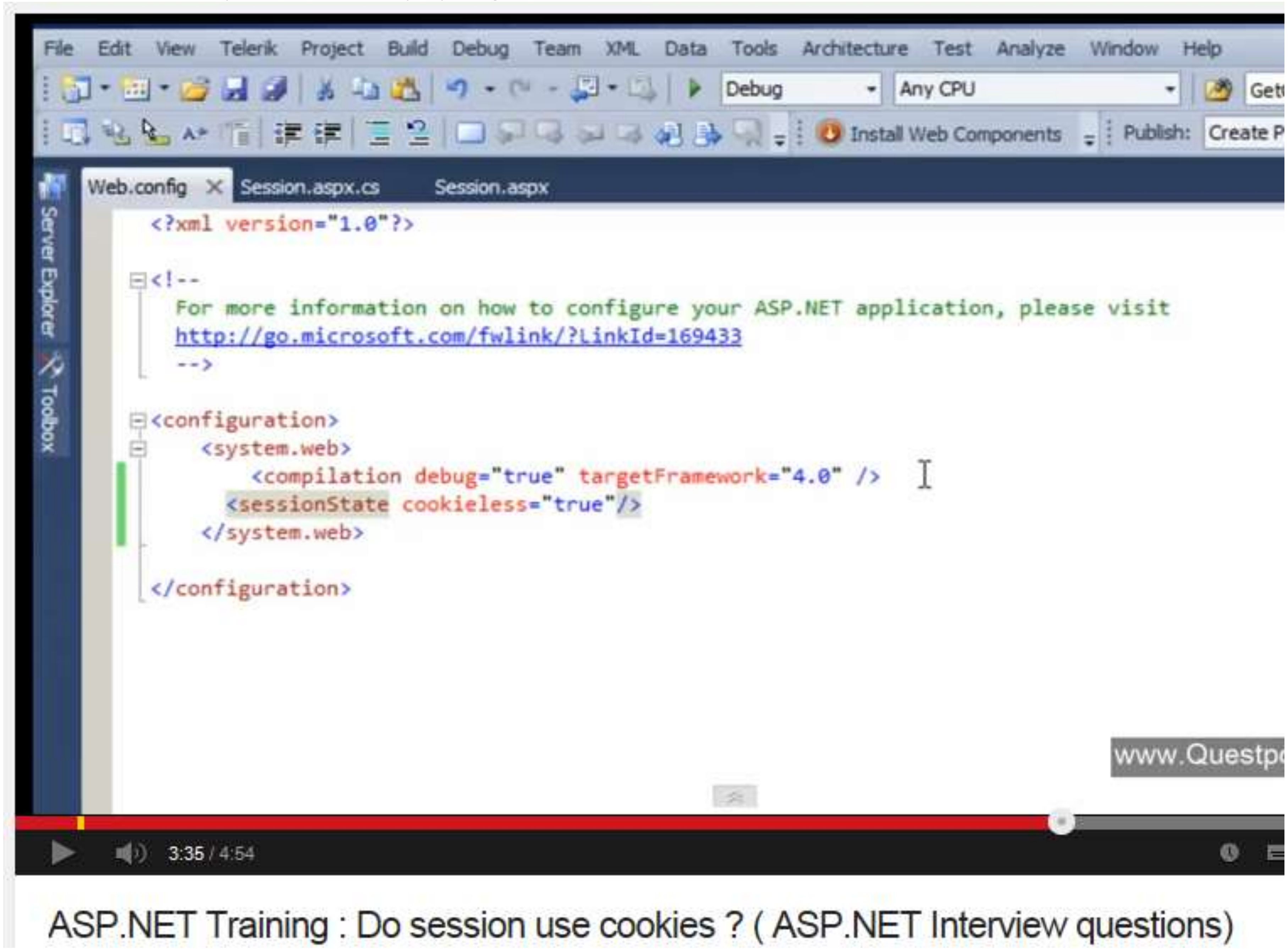


```
<?php
session_start();
session_destroy();
?>
```



```
<?php
session_start();

echo 'Welcome, '.$_SESSION['username'].'. You are '.$_SESSION['age'].'!';
```

If browser disable cookie, pass session info via query string.



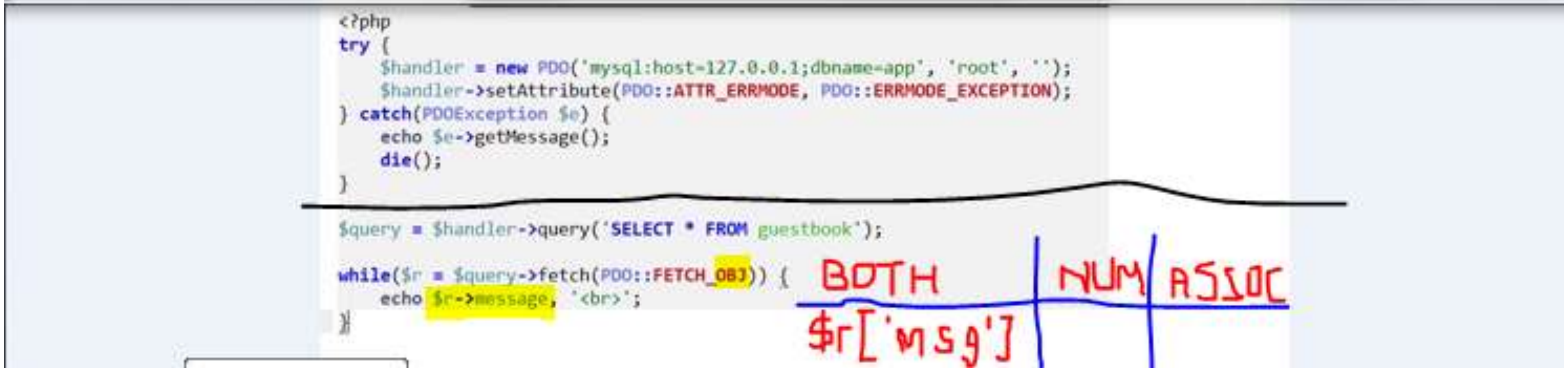
```
<?xml version="1.0"?>

<!--
  For more information on how to configure your ASP.NET application, please visit
  http://go.microsoft.com/fwlink/?LinkId=169433
  -->

<configuration>
    <system.web>
        <compilation debug="true" targetFramework="4.0" />
      <sessionState cookieless="true"/>
    </system.web>

</configuration>
```

ASP.NET Training : Do session use cookies ? (ASP.NET Interview questions)

PDO, and Prepared statements:(real escape string are obsolete)

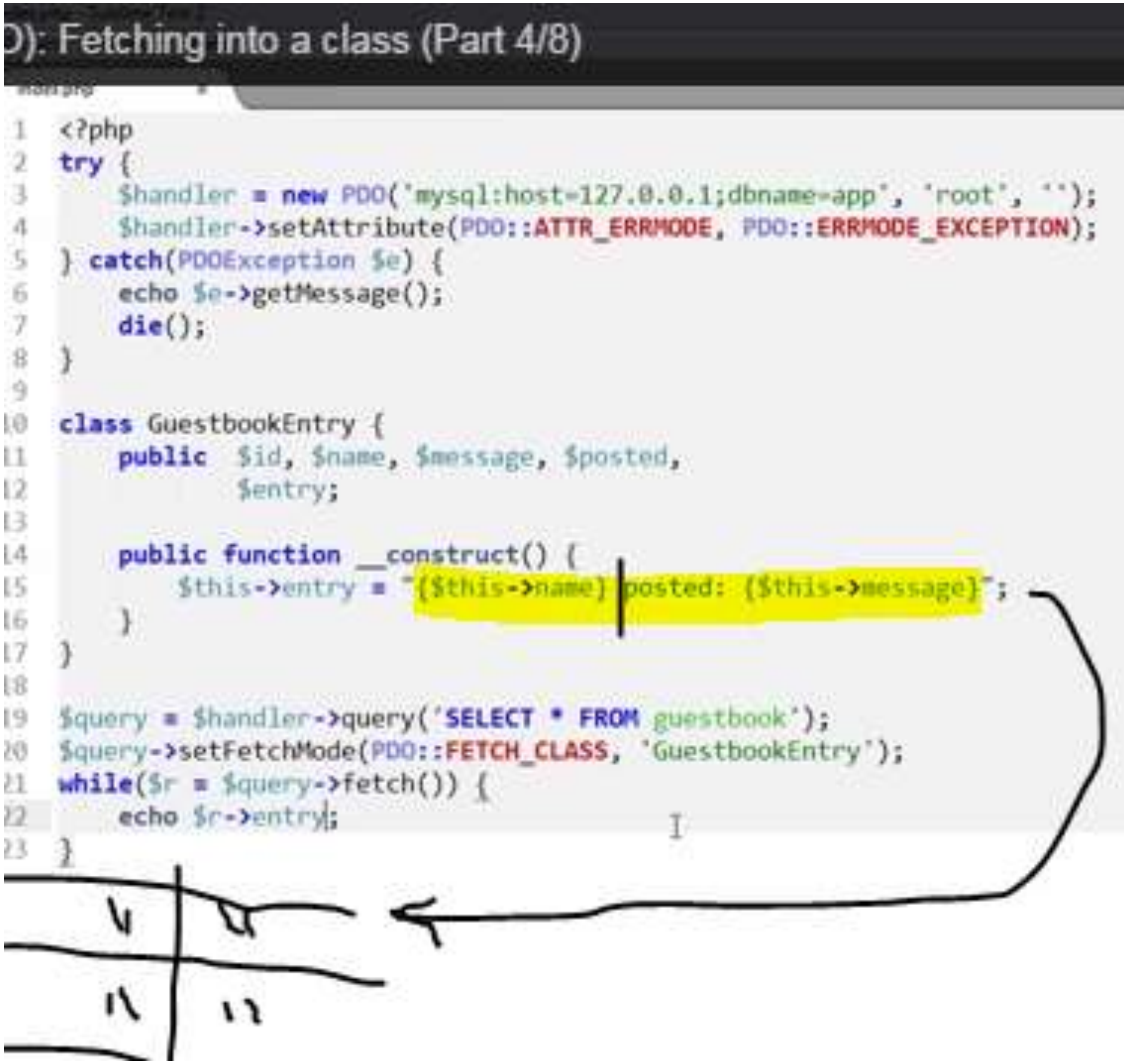
Handle error nicely AND Select regurgitate

```
print_r(PDO::getAvailableDrivers());     //find which db driver you have, eg mysql, sqlite...
```



```
<?php
try {
    $handler = new PDO('mysql:host=127.0.0.1;dbname=app', 'root', '');
    $handler->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch(PDOException $e) {
    echo $e->getMessage();
    die();
}

$query = $handler->query('SELECT * FROM guestbook');

while($r = $query->fetch(PDO::FETCH_OBJ)) {
    echo $r->message, '<br>';
}
```

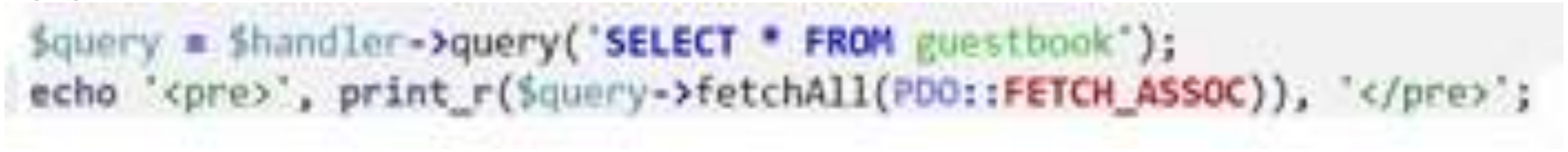
BOTH	NUM	ASSOC
$r['msg']		



D): Fetching into a class (Part 4/8)

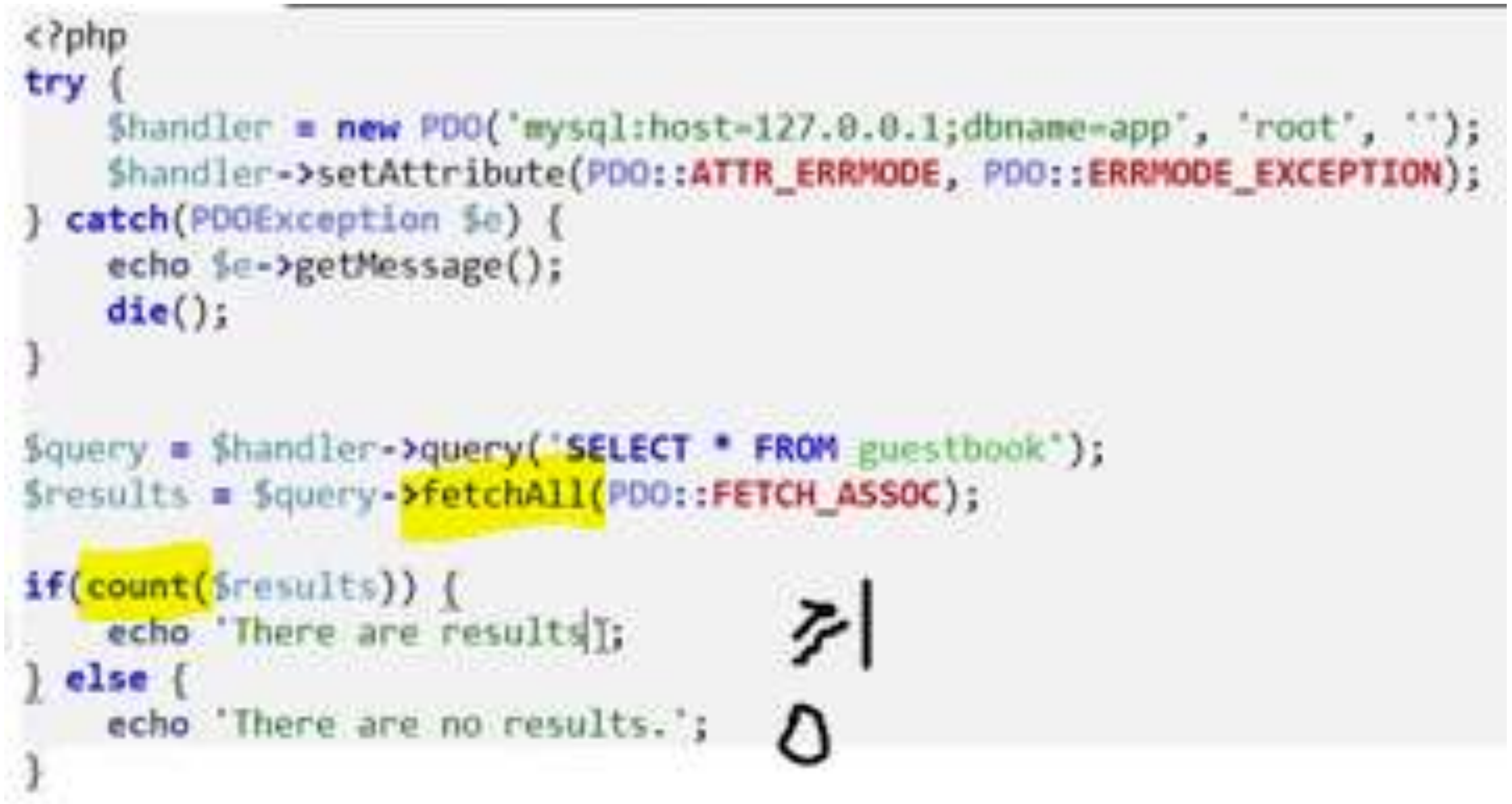
```php
<?php
try {
    $handler = new PDO('mysql:host=127.0.0.1;dbname=app', 'root', '');
    $handler->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch(PDOException $e) {
    echo $e->getMessage();
    die();
}

class GuestbookEntry {
    public  $id, $name, $message, $posted,
            $entry;

    public function __construct() {
        $this->entry = "{$this->name} posted: {$this->message}";
    }
}

$query = $handler->query('SELECT * FROM guestbook');
$query->setFetchMode(PDO::FETCH_CLASS, 'GuestbookEntry');
while($r = $query->fetch()) {
    echo $r->entry;
}
```

regurgitate all THE EASY WAY



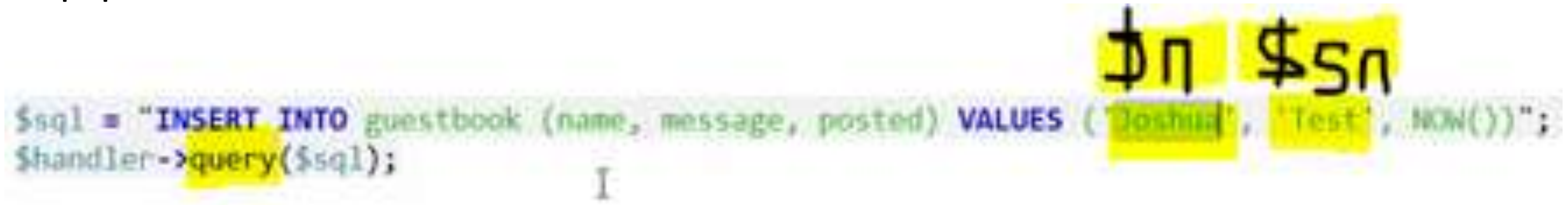
```
$query = $handler->query('SELECT * FROM guestbook');
echo '<pre>', print_r($query->fetchAll(PDO::FETCH_ASSOC)), '</pre>';
```

Count all



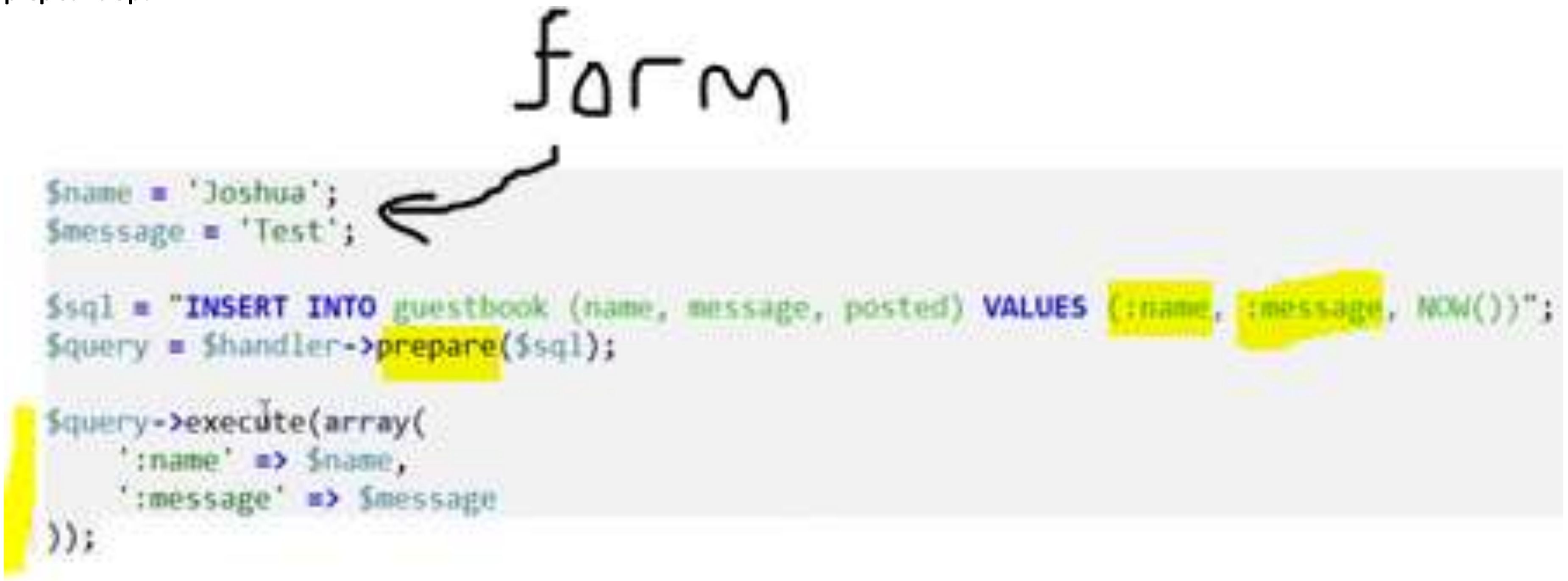
```
<?php
try {
    $handler = new PDO('mysql:host=127.0.0.1;dbname=app', 'root', '');
    $handler->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch(PDOException $e) {
    echo $e->getMessage();
    die();
}

$query = $handler->query('SELECT * FROM guestbook');
$results = $query->fetchAll(PDO::FETCH_ASSOC);

if(count($results)) {
    echo 'There are results';
} else {
    echo 'There are no results.';
}
```

Non prepared statement



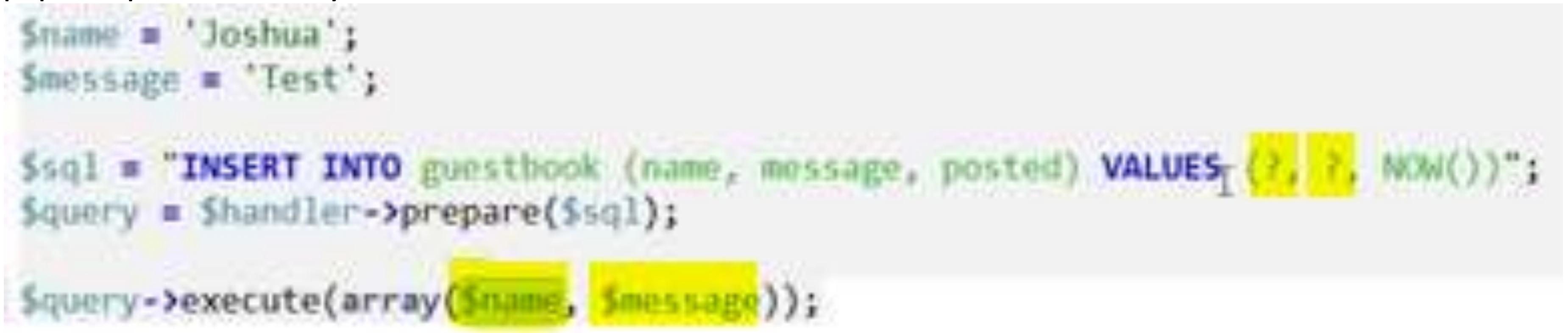
```
$sql = "INSERT INTO guestbook (name, message, posted) VALUES ('Joshua', 'Test', NOW())";
$handler->query($sql);
```

prep stmt Opt 1



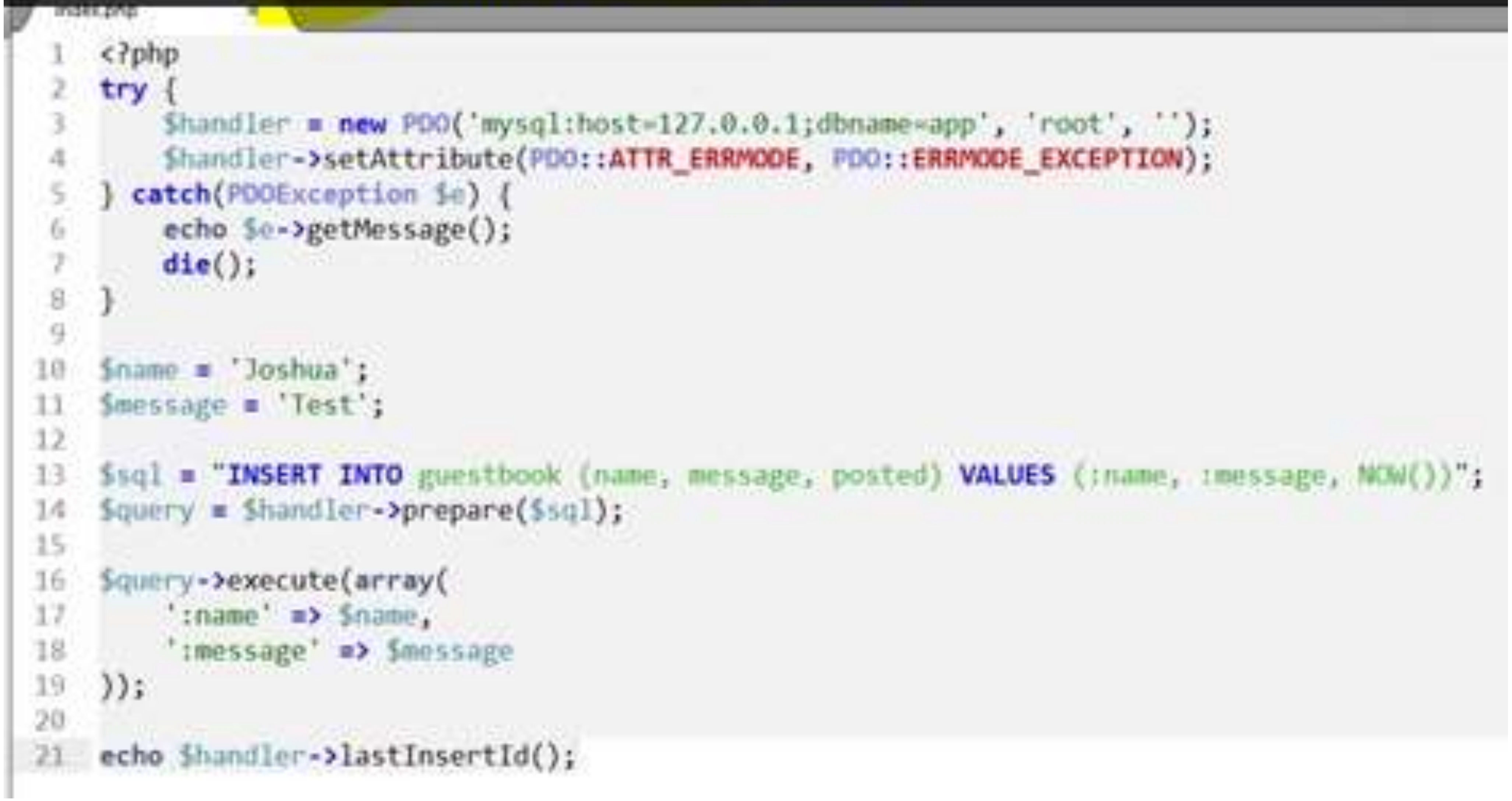
```php
$name = 'Joshua';
$message = 'Test';

$sql = "INSERT INTO guestbook (name, message, posted) VALUES (:name, :message, NOW())";
$query = $handler->prepare($sql);

$query->execute(array(
    ':name' => $name,
    ':message' => $message
));
```

prep stmt Opt 2 The shorter way



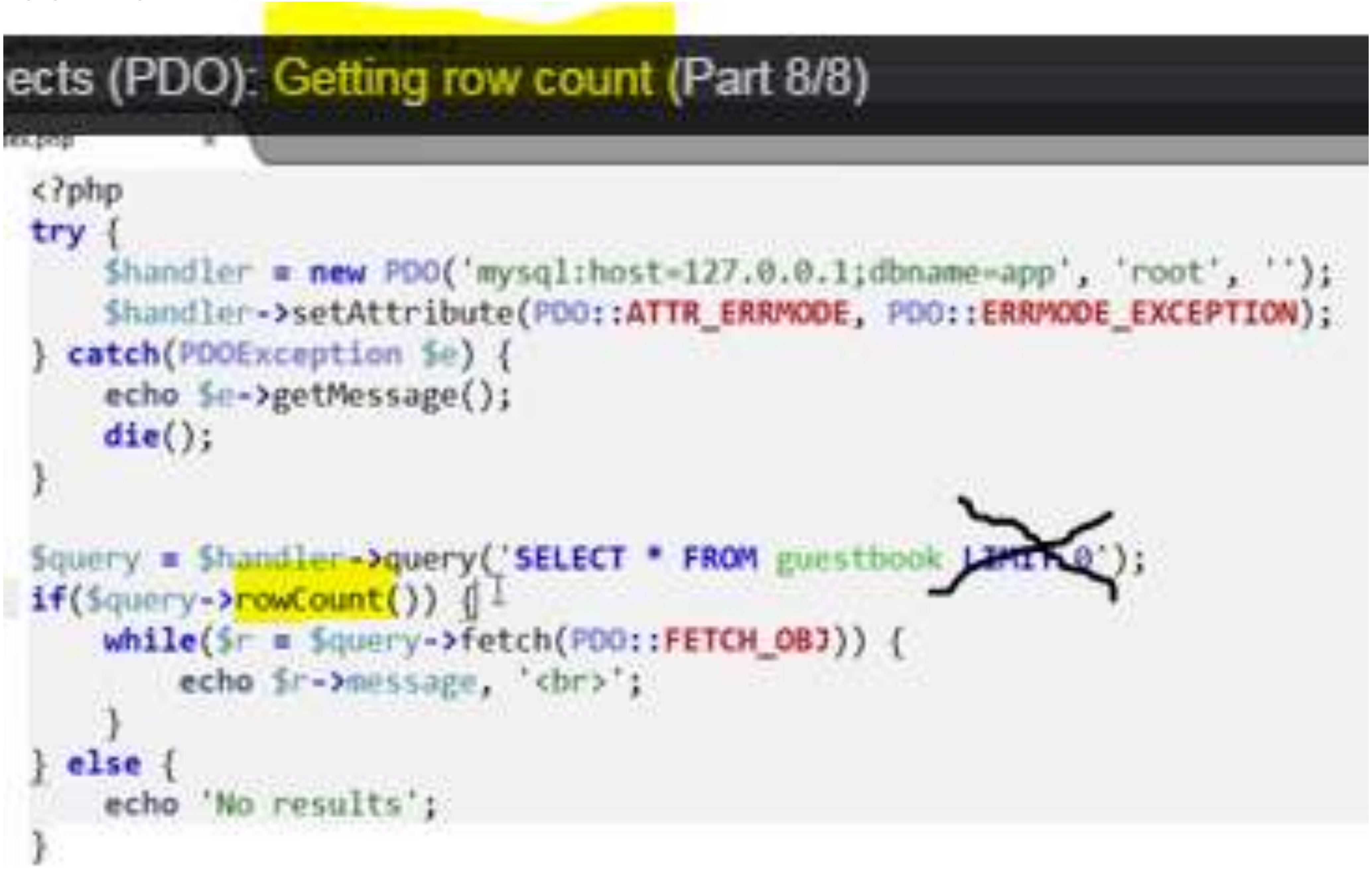
```php
$name = 'Joshua';
$message = 'Test';

$sql = "INSERT INTO guestbook (name, message, posted) VALUES (?, ?, NOW())";
$query = $handler->prepare($sql);

$query->execute(array($name, $message));
```

Objects (PDO): Last inserted ID (Part 7/8)


insert.php

```php
<?php
try {
    $handler = new PDO('mysql:host=127.0.0.1;dbname=app', 'root', '');
    $handler->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch(PDOException $e) {
    echo $e->getMessage();
    die();
}

$name = 'Joshua';
$message = 'Test';

$sql = "INSERT INTO guestbook (name, message, posted) VALUES (:name, :message, NOW())";
$query = $handler->prepare($sql);

$query->execute(array(
    ':name' => $name,
    ':message' => $message
));

echo $handler->lastInsertId();
```

Regurgitate using rowCount()

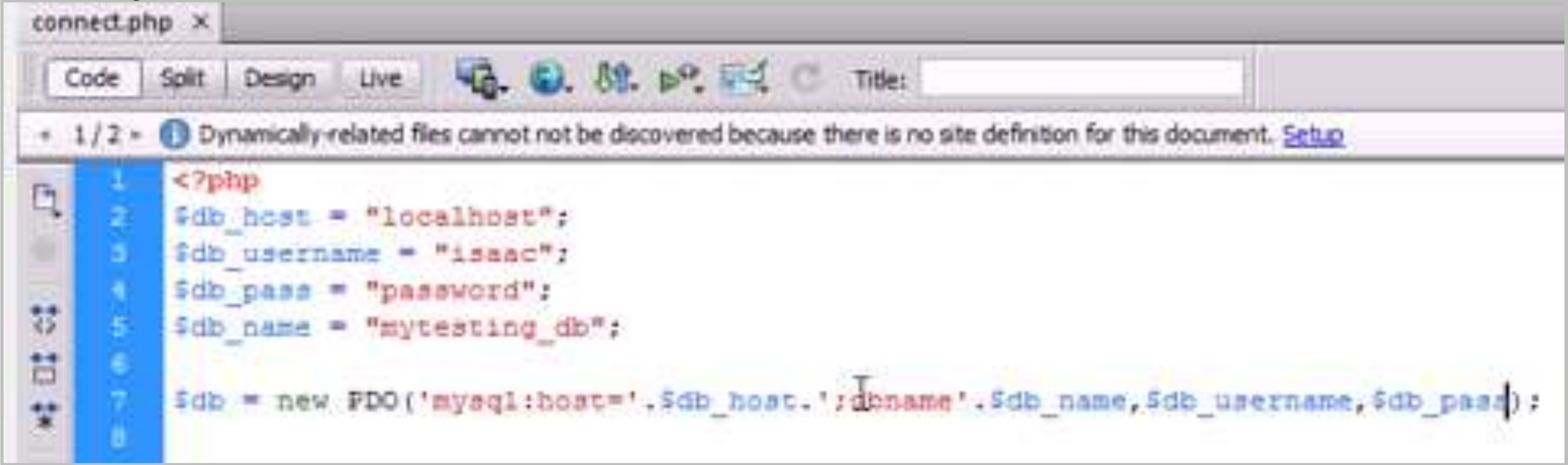


ects (PDO): Getting row count (Part 8/8)

```
<?php
try {
    $handler = new PDO('mysql:host=127.0.0.1;dbname=app', 'root', '');
    $handler->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch(PDOException $e) {
    echo $e->getMessage();
    die();
}

$query = $handler->query('SELECT * FROM guestbook ~~LIMIT 0~~');
if($query->rowCount()) {
    while($r = $query->fetch(PDO::FETCH_OBJ)) {
        echo $r->message, '<br>';
    }
} else {
    echo 'No results';
}
```

Unknown tutorial series

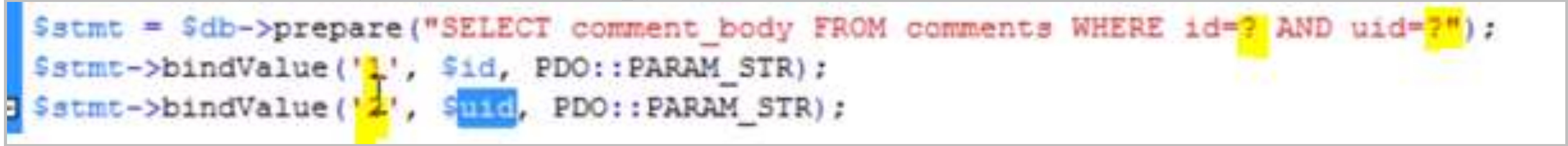
Connect Using PDO:



1 / 2 Dynamically-related files cannot not be discovered because there is no site definition for this document. Setup

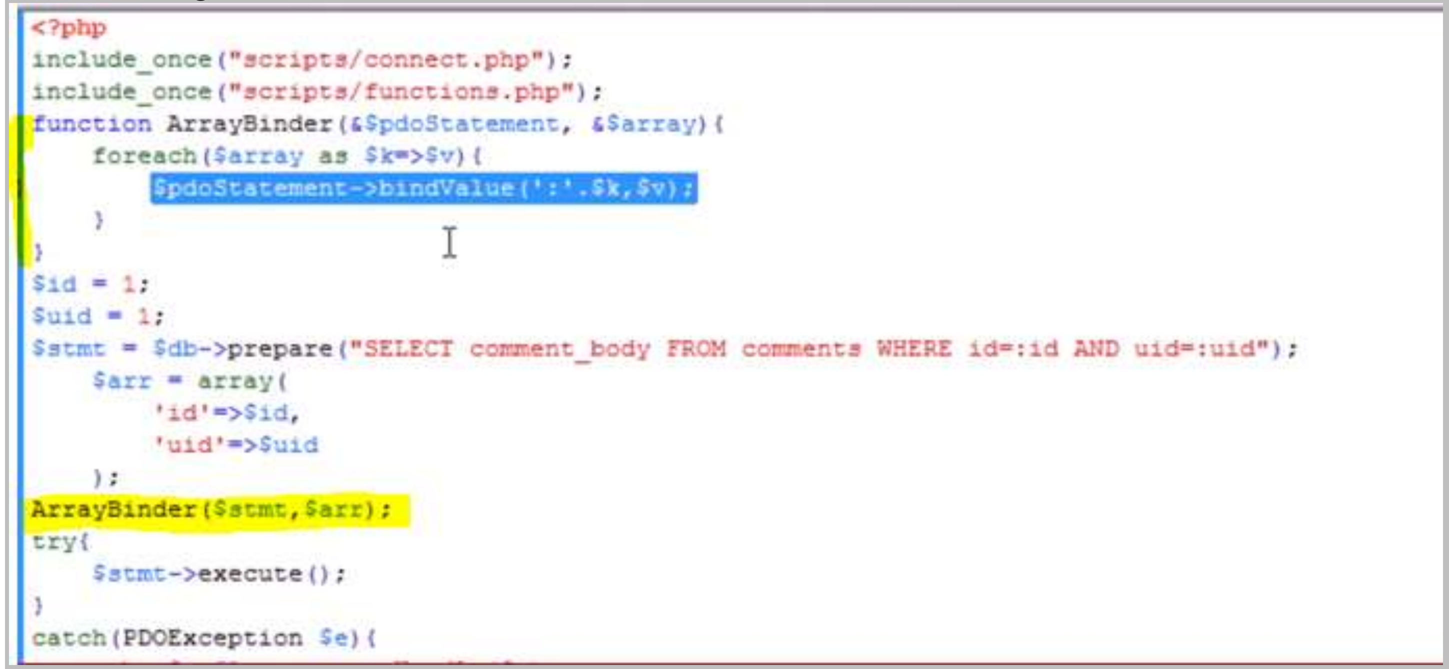
```
<?php
$db_host = "localhost";
$db_username = "isaac";
$db_pass = "password";
$db_name = "mytesting_db";

$db = new PDO('mysql:host='.$db_host.';dbname'.$db_name,$db_username,$db_pass);
```

bindvalue bind value to param, bind param bind param to value _ question mark means unamed values



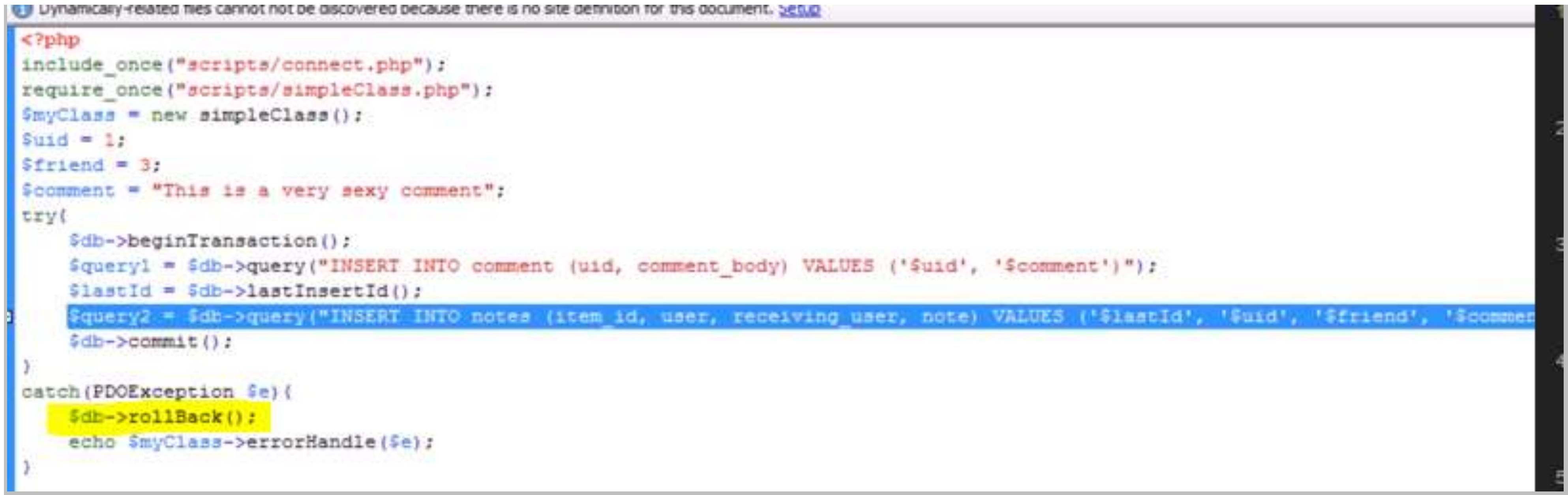
```
$stmt = $db->prepare("SELECT comment_body FROM comments WHERE id=? AND uid=?");
$stmt->bindValue('1', $id, PDO::PARAM_STR);
$stmt->bindValue('2', $uid, PDO::PARAM_STR);
```

Another binding variation



```
<?php
include_once("scripts/connect.php");
include_once("scripts/functions.php");
function ArrayBinder(&$pdoStatement, &$array){
    foreach($array as $k=>$v){
        $pdoStatement->bindValue(':'.$k,$v);
    }
}
$id = 1;
$uid = 1;
$stmt = $db->prepare("SELECT comment_body FROM comments WHERE id=:id AND uid=:uid");
    $arr = array(
        'id'=>$id,
        'uid'=>$uid
    );
ArrayBinder($stmt,$arr);
try{
    $stmt->execute();
}
catch(PDOException $e){
```

rollback ensures that either none or all the sql queries in one try block gets made

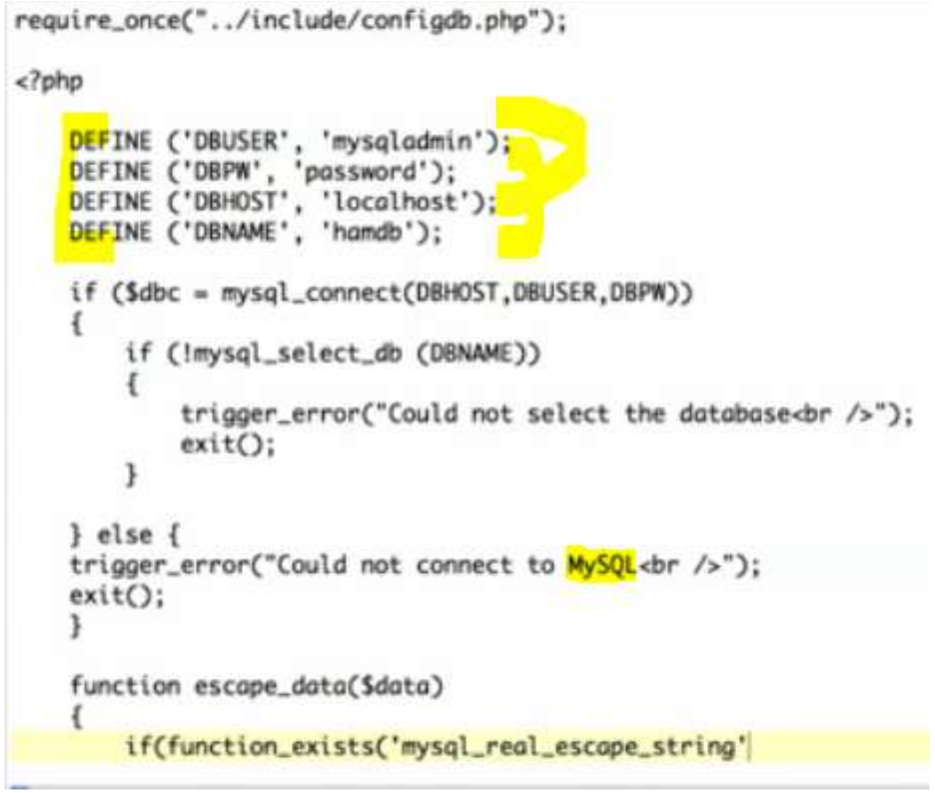


Dynamically-related files cannot not be discovered because there is no site definition for this document. Setup

```
<?php
include_once("scripts/connect.php");
require_once("scripts/simpleClass.php");
$myClass = new simpleClass();
$uid = 1;
$friend = 3;
$comment = "This is a very sexy comment";
try{
    $db->beginTransaction();
    $query1 = $db->query("INSERT INTO comment (uid, comment_body) VALUES ('$uid', '$comment')");
    $lastId = $db->lastInsertId();
    $query2 = $db->query("INSERT INTO notes (item_id, user, receiving_user, note) VALUES ('$lastId', '$uid', '$friend', '$commen
    $db->commit();
}
catch(PDOException $e){
    $db->rollBack();
    echo $myClass->errorHandle($e);
}
```

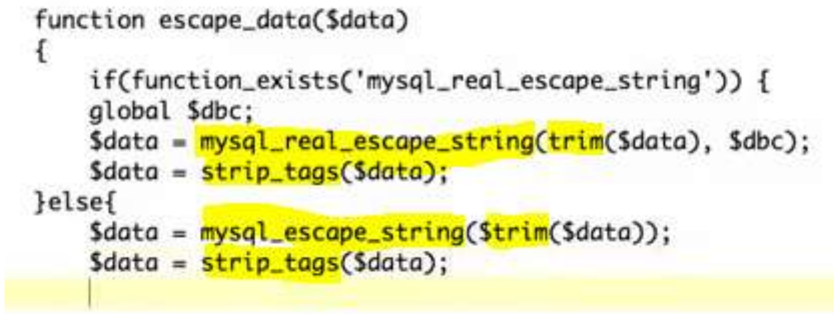
Banas Tutorials

PHP SECU 1 - use constants to compose conn str, make your config less guessable and intuitive, dont tell them honestly that ur using mysql



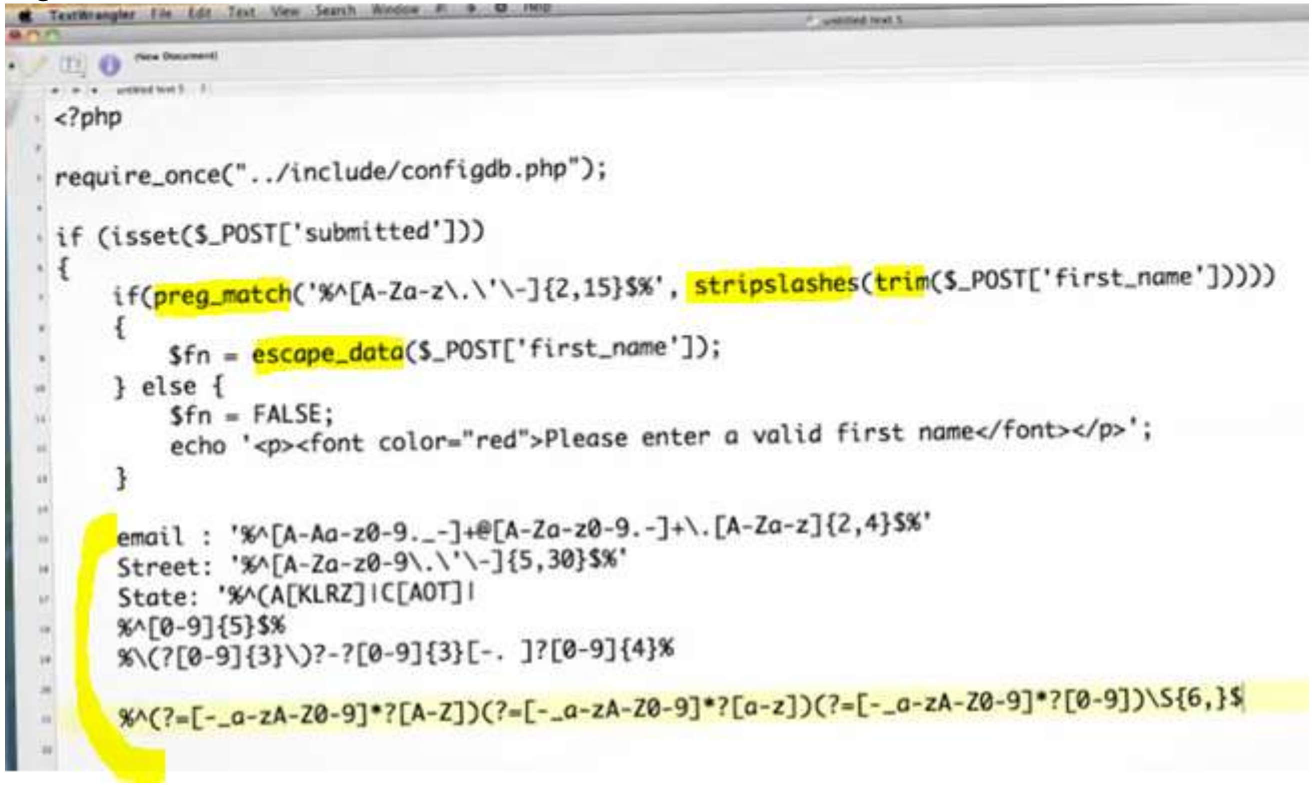
```
require_once("../include/configdb.php");

<?php

    DEFINE ('DBUSER', 'mysqladmin');
    DEFINE ('DBPW', 'password');
    DEFINE ('DBHOST', 'localhost');
    DEFINE ('DBNAME', 'hamdb');

    if ($dbc = mysql_connect(DBHOST,DBUSER,DBPW))
    {
        if (!mysql_select_db (DBNAME))
        {
            trigger_error("Could not select the database<br />");
            exit();
        }

    } else {
    trigger_error("Could not connect to MySQL<br />");
    exit();
    }

    function escape_data($data)
    {
        if(function_exists('mysql_real_escape_string'
```

Escaping



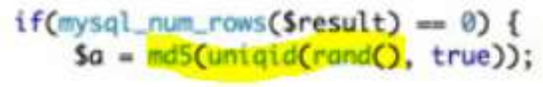
```
 function escape_data($data)
 {
     if(function_exists('mysql_real_escape_string')) {
     global $dbc;
     $data = mysql_real_escape_string(trim($data), $dbc);
     $data = strip_tags($data);
 }else{
     $data = mysql_escape_string($trim($data));
     $data = strip_tags($data);
```

Regex in backend



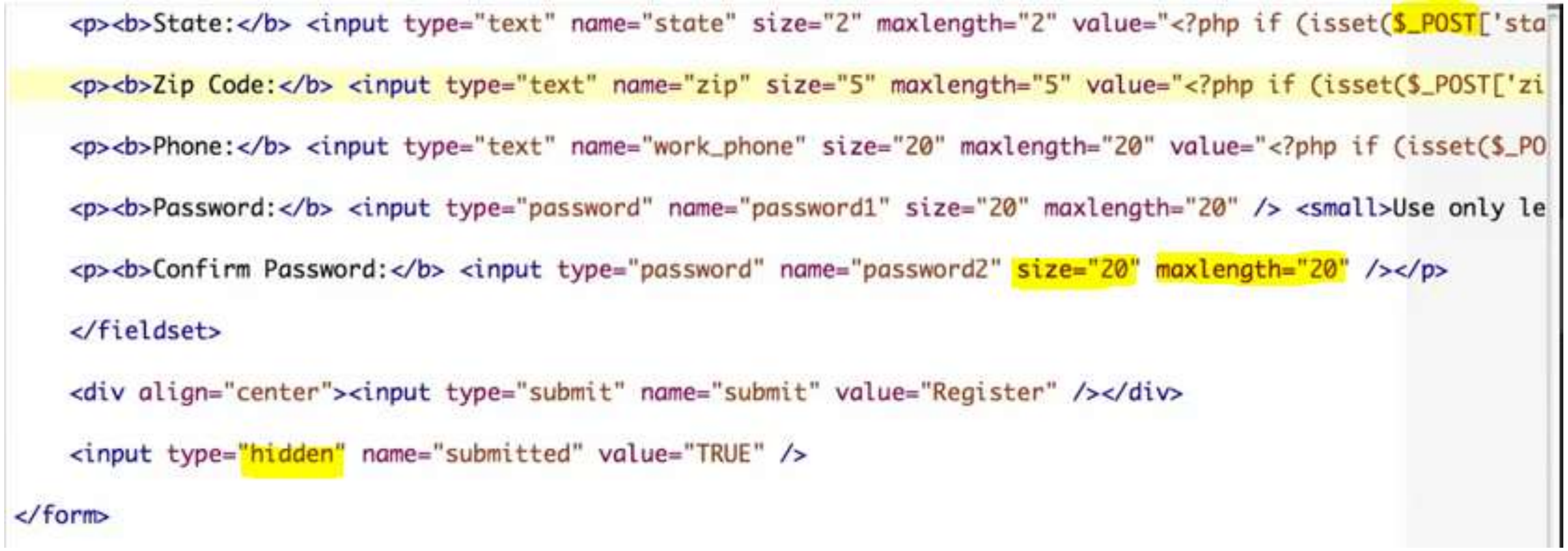
```
<?php

require_once("../include/configdb.php");

if (isset($_POST['submitted']))
{
    if(preg_match('%^[A-Za-z\.\'\-]{2,15}$%', stripslashes(trim($_POST['first_name']))))
    {
        $fn = escape_data($_POST['first_name']);
    } else {
        $fn = FALSE;
        echo '<p><font color="red">Please enter a valid first name</font></p>';
    }

    email : '%^[A-Aa-z0-9._-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}$%'
    Street: '%^[A-Za-z0-9\.\'\-]{5,30}$%'
    State: '%^(A[KLRZ]|C[AOT]|
    %^[0-9]{5}$%
    %\(?[0-9]{3}\)?-?[0-9]{3}[-. ]?[0-9]{4}%

    %^(?=[-_a-zA-Z0-9]*?[A-Z])(?=[-_a-zA-Z0-9]*?[a-z])(?=[-_a-zA-Z0-9]*?[0-9])\S{6,}$
```

PHP SECU 3 - activition code _ dont use multi query, just exec 1 query at a timegenerated by 3 random-gen funcs

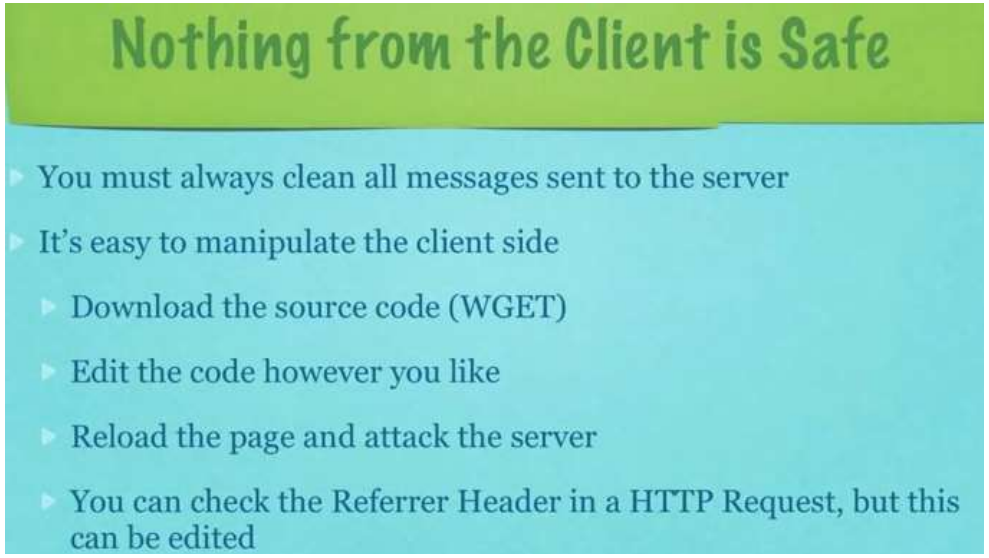


```
if(mysql_num_rows($result) == 0) {
    $a = md5(uniqid(rand(), true));
```

PHP SECU 4 - frontend form secu _ in backend sql inserts, SHA the pw before putting pw into db

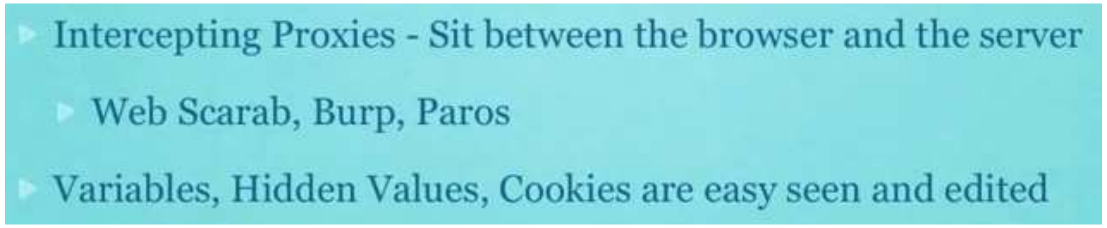


```
    <p><b>State:</b> <input type="text" name="state" size="2" maxlength="2" value="<?php if (isset($_POST['sta

    <p><b>Zip Code:</b> <input type="text" name="zip" size="5" maxlength="5" value="<?php if (isset($_POST['zi

    <p><b>Phone:</b> <input type="text" name="work_phone" size="20" maxlength="20" value="<?php if (isset($_PO

    <p><b>Password:</b> <input type="password" name="password1" size="20" maxlength="20" /> <small>Use only le

    <p><b>Confirm Password:</b> <input type="password" name="password2" size="20" maxlength="20" /></p>

    </fieldset>

    <div align="center"><input type="submit" name="submit" value="Register" /></div>

    <input type="hidden" name="submitted" value="TRUE" />

</form>
```

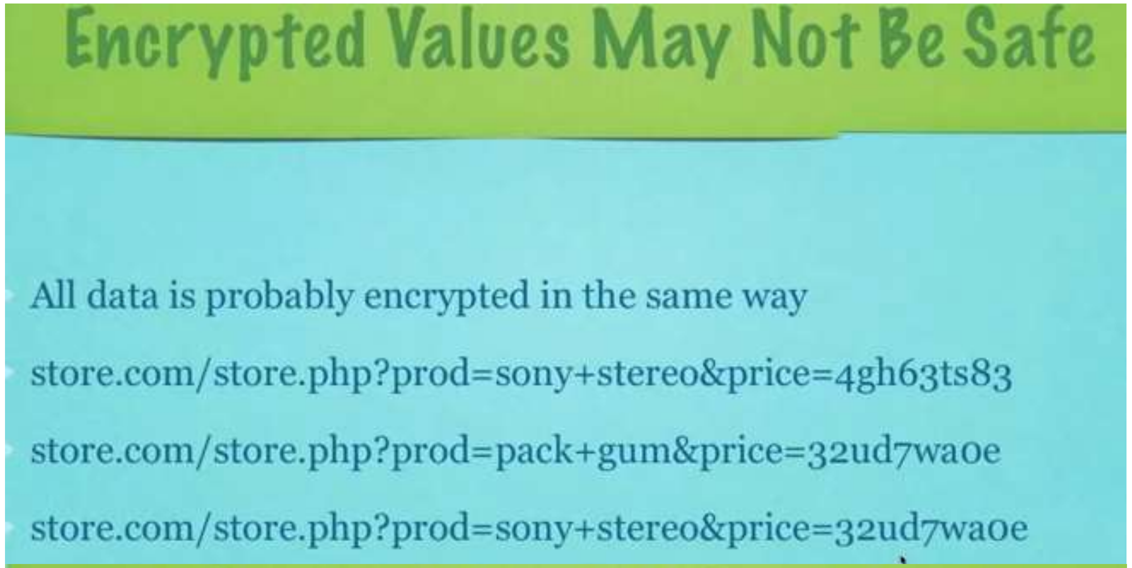


Nothing from the Client is Safe

- You must always clean all messages sent to the server
- It's easy to manipulate the client side
  - Download the source code (WGET)
  - Edit the code however you like
  - Reload the page and attack the server
  - You can check the Referrer Header in a HTTP Request, but this can be edited

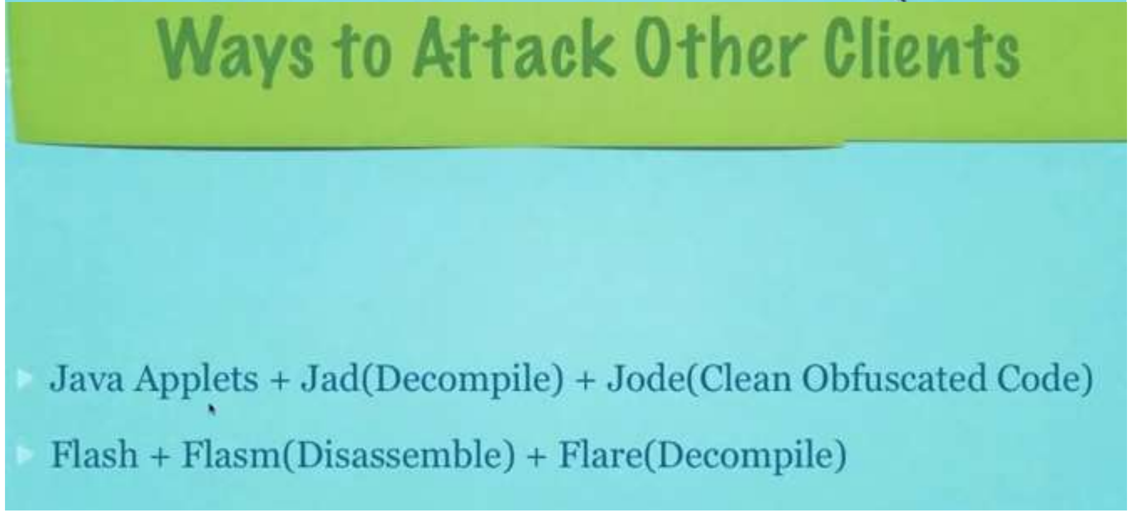


- Intercepting Proxies - Sit between the browser and the server
  - Web Scarab, Burp, Paros
- Variables, Hidden Values, Cookies are easy seen and edited



Encrypted Values May Not Be Safe

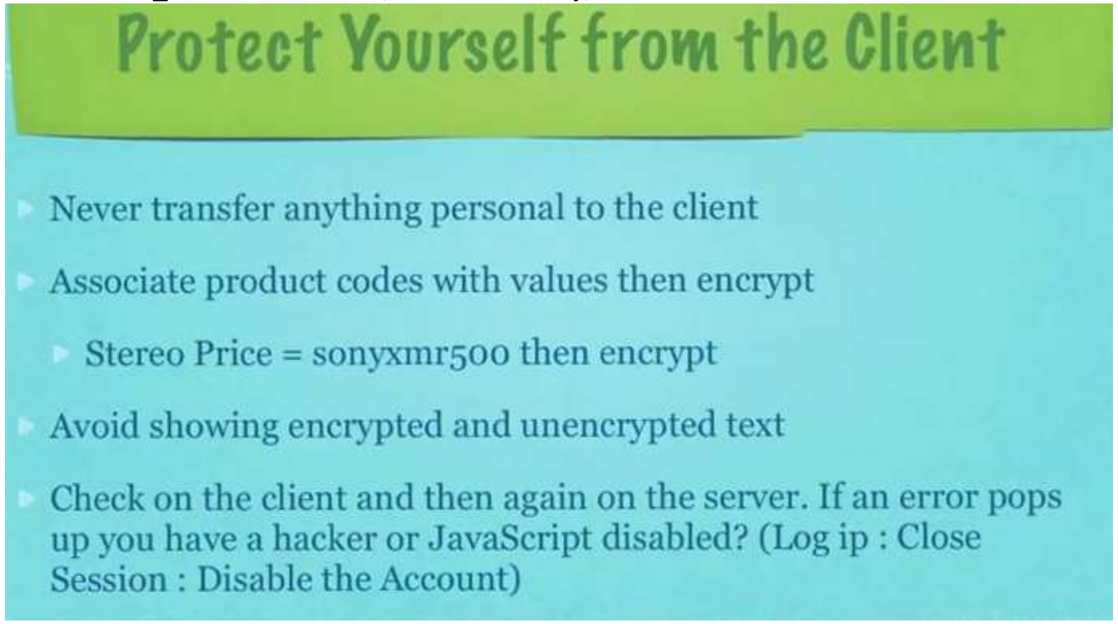
- All data is probably encrypted in the same way
- store.com/store.php?prod=sony+stereo&price=4gh63ts83
- store.com/store.php?prod=pack+gum&price=32ud7wa0e
- store.com/store.php?prod=sony+stereo&price=32ud7wa0e



Ways to Attack Other Clients

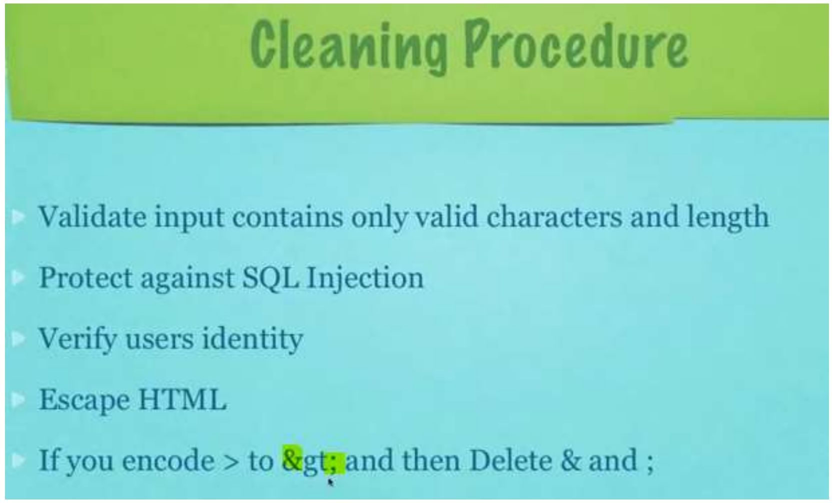
- Java Applets + Jad(Decompile) + Jode(Clean Obfuscated Code)
- Flash + Flasm(Disassemble) + Flare(Decompile)

PHP SECU 8 _ secu cli n svr side, if inconsistency, then del their account



Protect Yourself from the Client

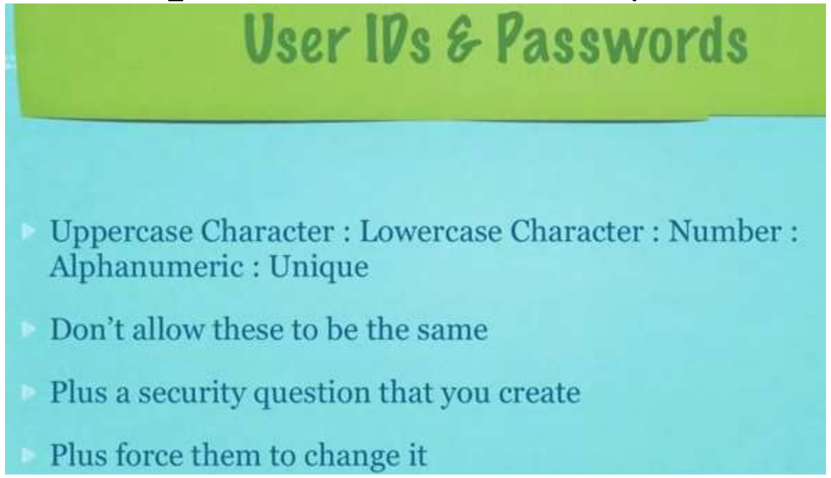
- Never transfer anything personal to the client
- Associate product codes with values then encrypt
  - Stereo Price = sonyxmr500 then encrypt
- Avoid showing encrypted and unencrypted text
- Check on the client and then again on the server. If an error pops up you have a hacker or JavaScript disabled? (Log ip : Close Session : Disable the Account)



Cleaning Procedure

- Validate input contains only valid characters and length
- Protect against SQL Injection
- Verify users identity
- Escape HTML
- If you encode > to &gt; and then Delete & and ;

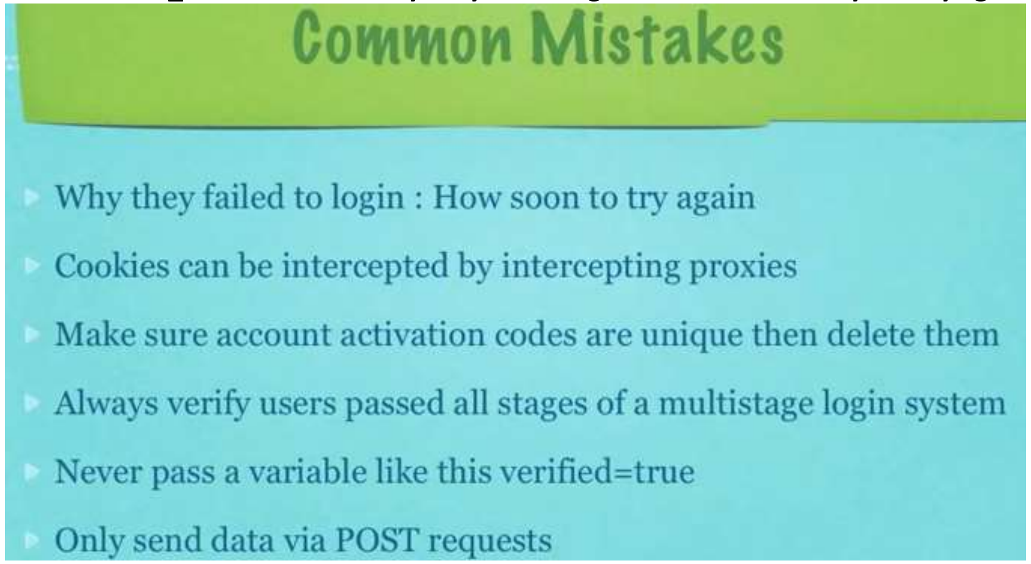
PHP SECU 10 _ dont let them make their own secu q



User IDs & Passwords

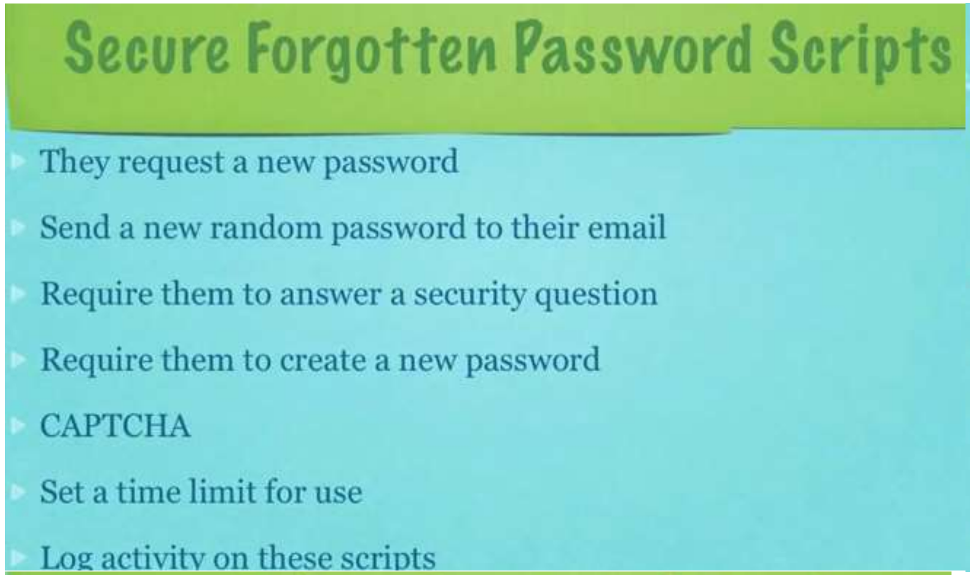
- Uppercase Character : Lowercase Character : Number : Alphanumeric : Unique
- Don't allow these to be the same
- Plus a security question that you create
- Plus force them to change it

PHP SECU 11 _ dont tell them why they failed login and how soon they can try again



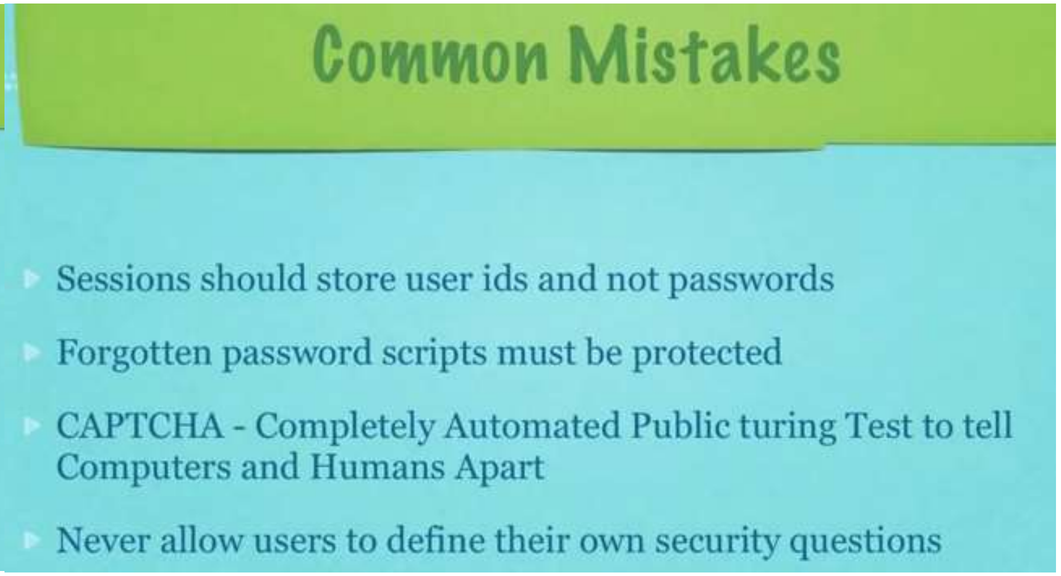
Common Mistakes

- Why they failed to login : How soon to try again
- Cookies can be intercepted by intercepting proxies
- Make sure account activation codes are unique then delete them
- Always verify users passed all stages of a multistage login system
- Never pass a variable like this verified=true
- Only send data via POST requests



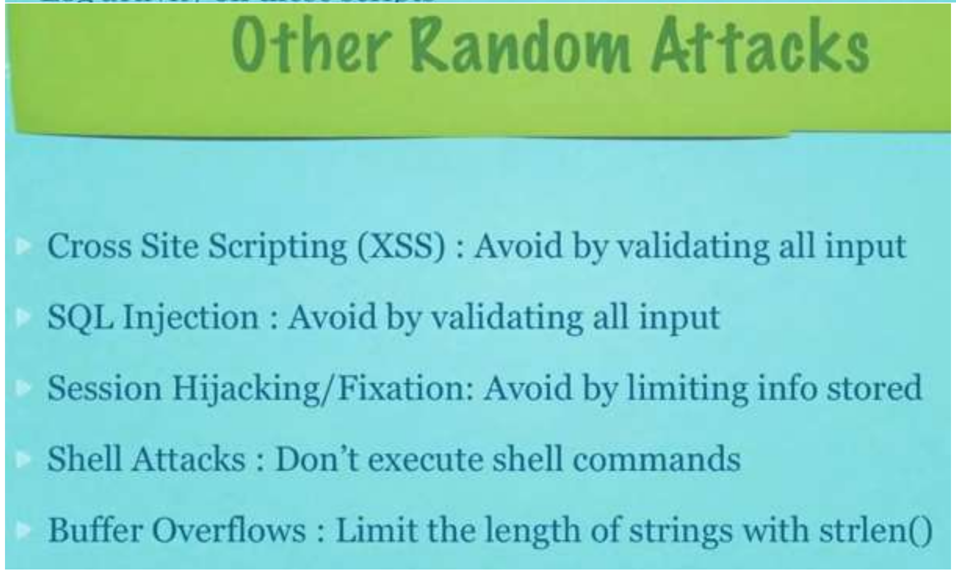
Secure Forgotten Password Scripts

- They request a new password
- Send a new random password to their email
- Require them to answer a security question
- Require them to create a new password
- CAPTCHA
- Set a time limit for use
- Log activity on these scripts



Common Mistakes

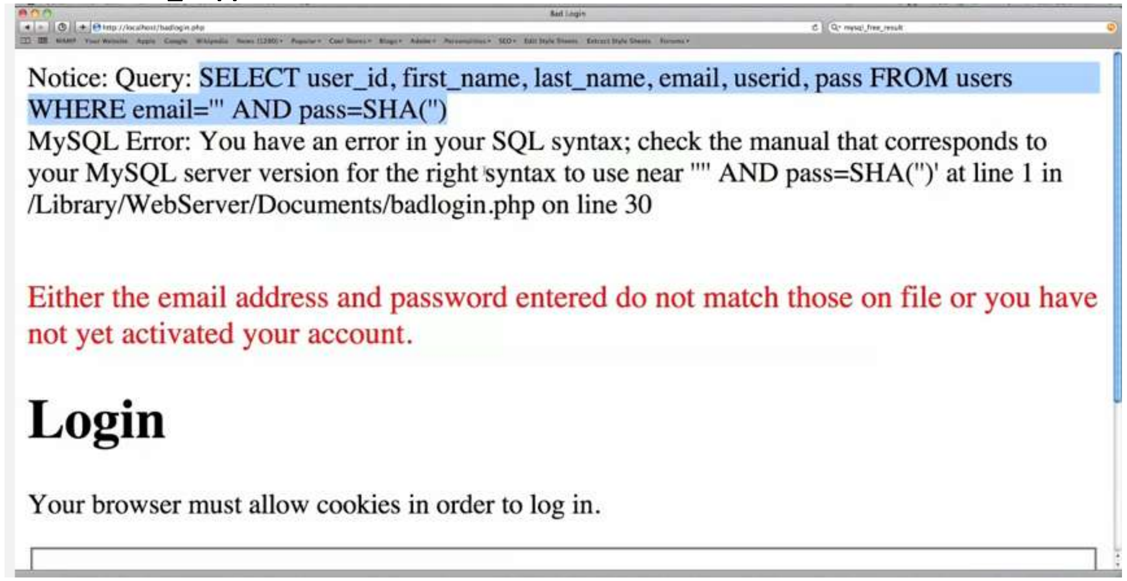
- Sessions should store user ids and not passwords
- Forgotten password scripts must be protected
- CAPTCHA - Completely Automated Public turing Test to tell Computers and Humans Apart
- Never allow users to define their own security questions



Other Random Attacks

- Cross Site Scripting (XSS) : Avoid by validating all input
- SQL Injection : Avoid by validating all input
- Session Hijacking/Fixation: Avoid by limiting info stored
- Shell Attacks : Don’t execute shell commands
- Buffer Overflows : Limit the length of strings with strlen()

PHP SECU 15 _ suppress errors on site

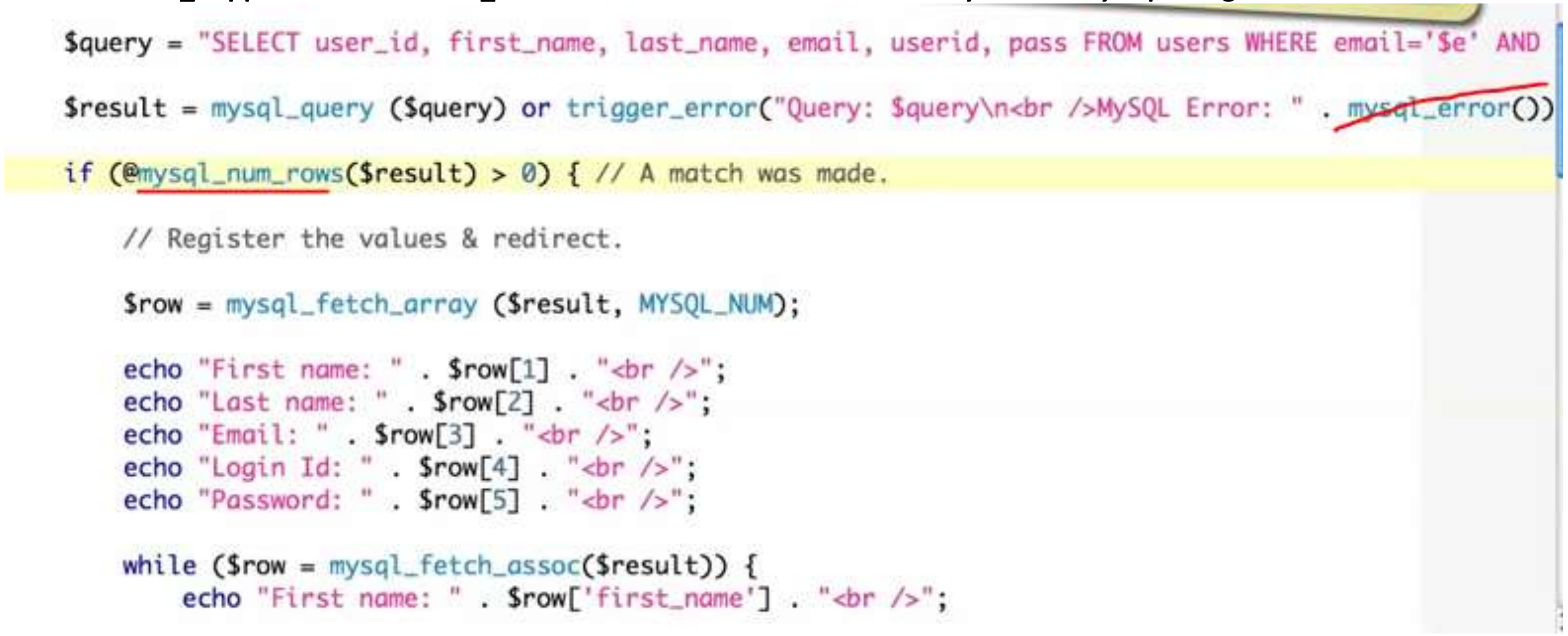

Notice: Query: SELECT user_id, first_name, last_name, email, userid, pass FROM users WHERE email='' AND pass=SHA('')
MySQL Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near ''' AND pass=SHA('')' at line 1 in /Library/WebServer/Documents/badlogin.php on line 30

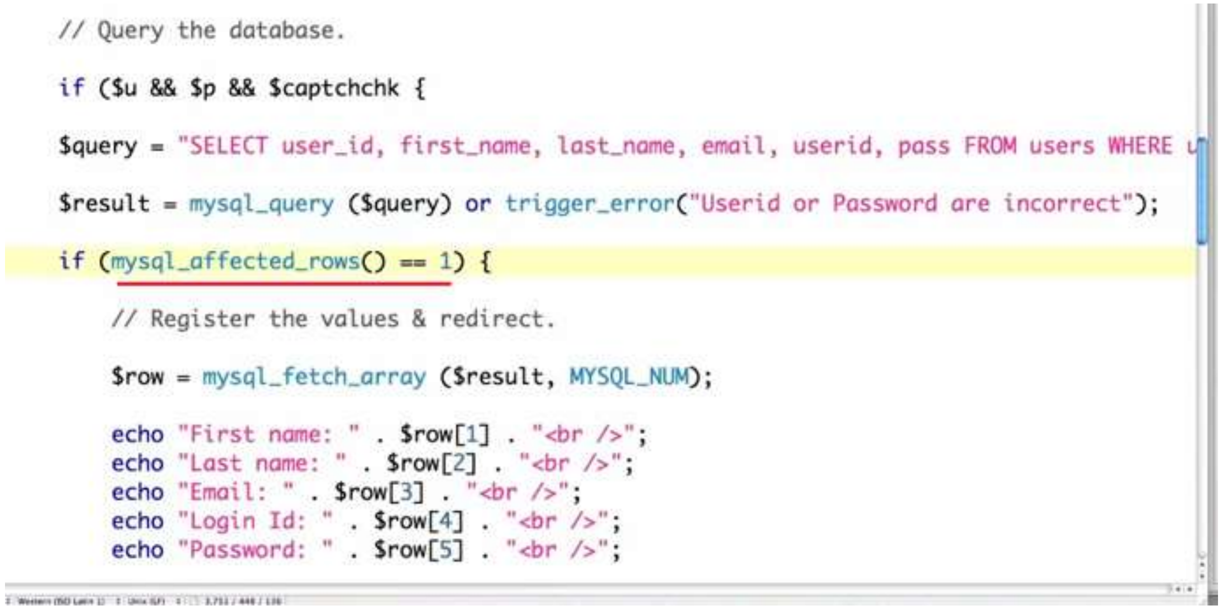
Either the email address and password entered do not match those on file or you have not yet activated your account.

Login

Your browser must allow cookies in order to log in.

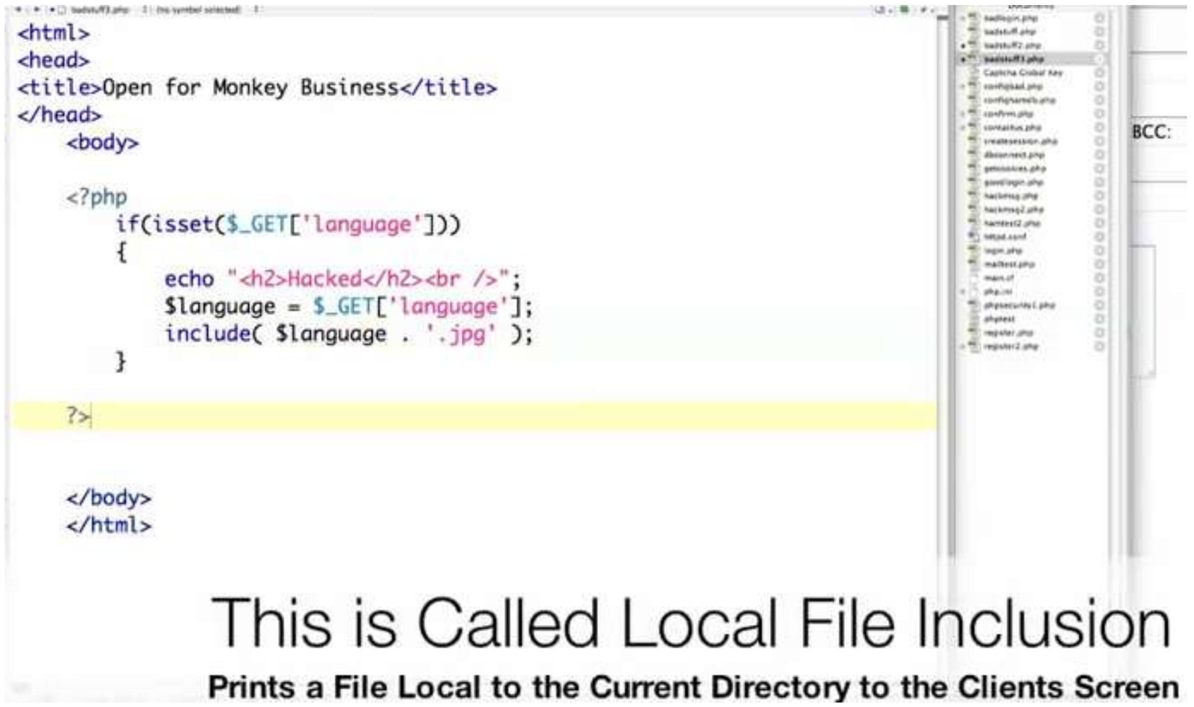
PHP SECU 15a _ suppress errors on site _ use affected rows EQUALS 1 instead if you are only expecting 1 rcd in ret


```php
$query = "SELECT user_id, first_name, last_name, email, userid, pass FROM users WHERE email='$e' AND

$result = mysql_query ($query) or trigger_error("Query: $query\n<br />MySQL Error: " . ~~mysql_error~~())

if (@mysql_num_rows($result) > 0) { // A match was made.

    // Register the values & redirect.

    $row = mysql_fetch_array ($result, MYSQL_NUM);

    echo "First name: " . $row[1] . "<br />";
    echo "Last name: " . $row[2] . "<br />";
    echo "Email: " . $row[3] . "<br />";
    echo "Login Id: " . $row[4] . "<br />";
    echo "Password: " . $row[5] . "<br />";

    while ($row = mysql_fetch_assoc($result)) {
        echo "First name: " . $row['first_name'] . "<br />";
```



```
// Query the database.

if ($u && $p && $captchchk {

$query = "SELECT user_id, first_name, last_name, email, userid, pass FROM users WHERE u

$result = mysql_query ($query) or trigger_error("Userid or Password are incorrect");

if (mysql_affected_rows() == 1) {

    // Register the values & redirect.

    $row = mysql_fetch_array ($result, MYSQL_NUM);

    echo "First name: " . $row[1] . "<br />";
    echo "Last name: " . $row[2] . "<br />";
    echo "Email: " . $row[3] . "<br />";
    echo "Login Id: " . $row[4] . "<br />";
    echo "Password: " . $row[5] . "<br />";
```

File inclusion

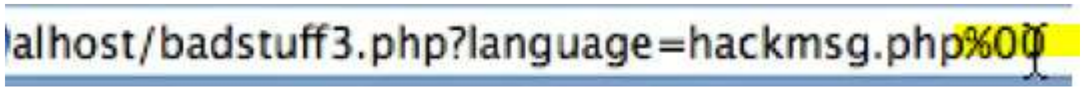


```
<html>
<head>
<title>Open for Monkey Business</title>
</head>
    <body>

    <?php
        if(isset($_GET['language']))
        {
            echo "<h2>Hacked</h2><br />";
            $language = $_GET['language'];
            include( $language . '.jpg' );
        }

    ?>


    </body>
    </html>
```

This is Called Local File Inclusion

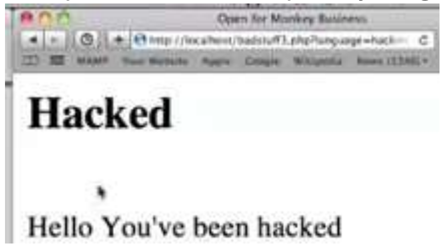
Prints a File Local to the Current Directory to the Clients Screen

16b : %00 nullified (makes ineffective) everything after.



```
alhost/badstuff3.php?language=hackmsg.php%00
```

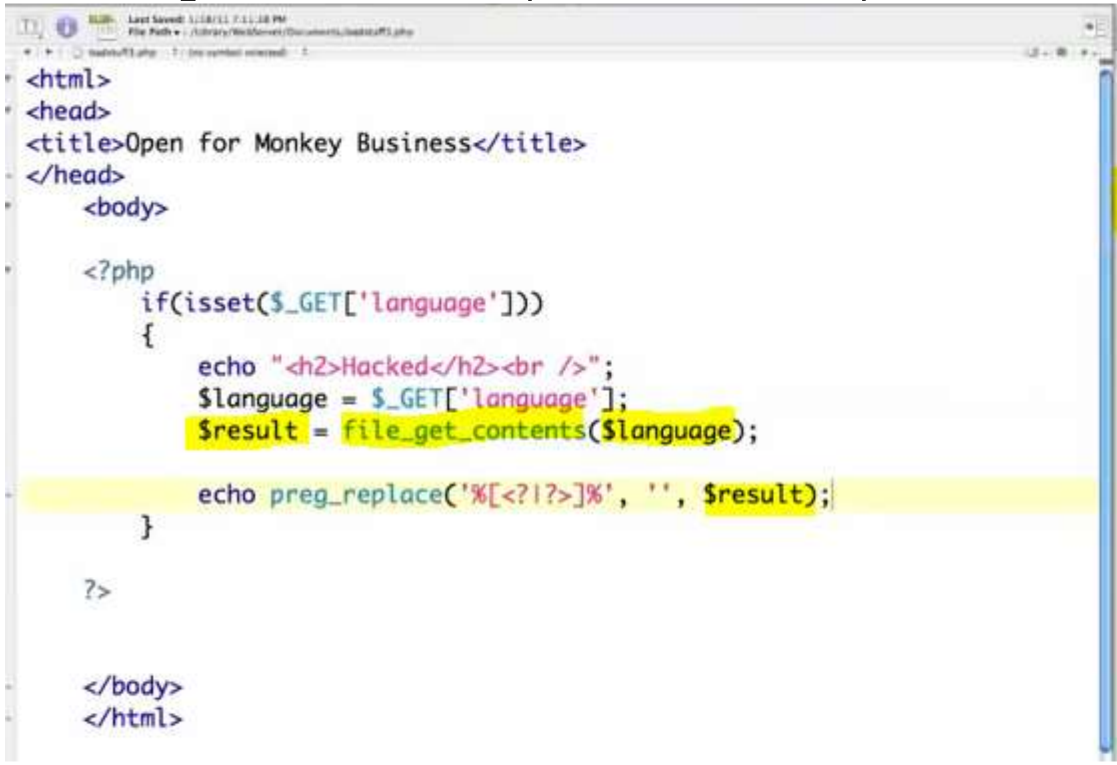
Result of 16b:



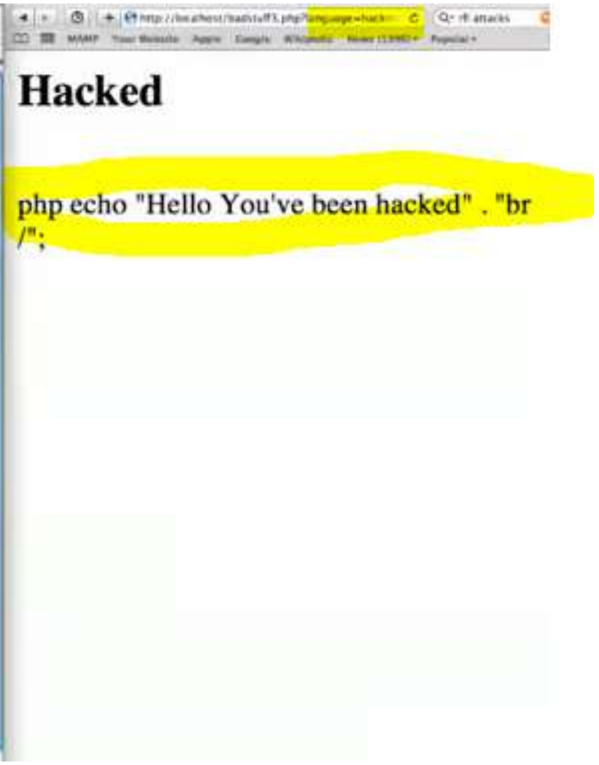
Hacked

Hello You've been hacked

PHP SECU 16c _ VARIATION of 16b, hacker print the content of files in your svr



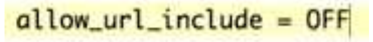
```
<html>
<head>
<title>Open for Monkey Business</title>
</head>
    <body>

    <?php
        if(isset($_GET['language']))
        {
            echo "<h2>Hacked</h2><br />";
            $language = $_GET['language'];
            $result = file_get_contents($language);

            echo preg_replace('%[<?|?>]%', '', $result);
        }

    ?>


    </body>
    </html>
```



Hacked

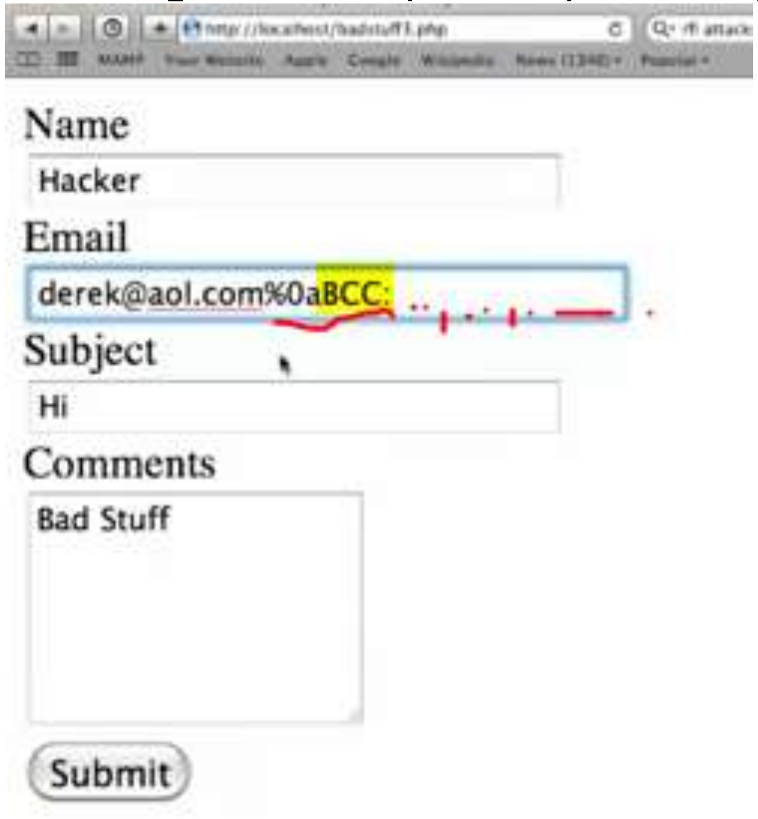
php echo "Hello You've been hacked" . "br
/";

PHP SECU 16d _ prevent 16b scenario by using this in phpINI file



```
allow_url_include = OFF
```

PHP SECU 17 _ make server mail you and many other BCC recipients of your chosing, if svr uses mail method directly



Name

Hacker

Email

derek@aol.com%0aBCC:

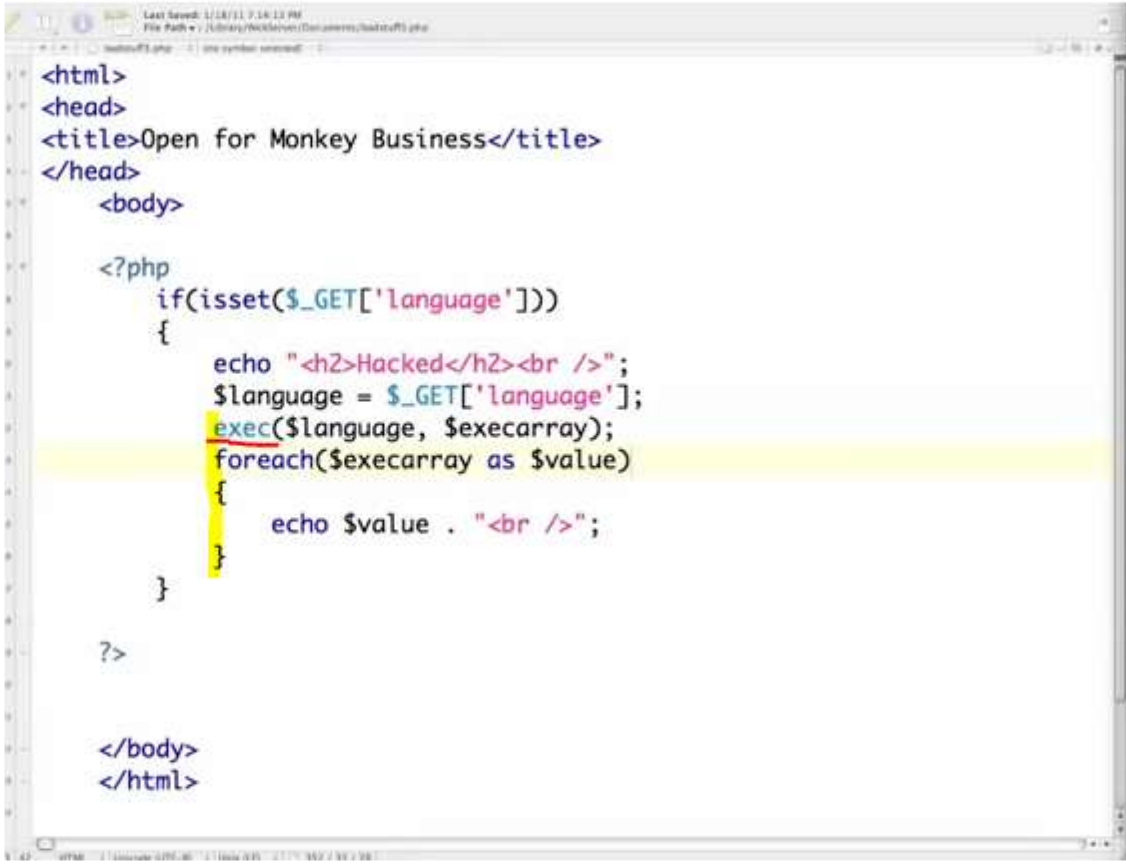
Subject

Hi

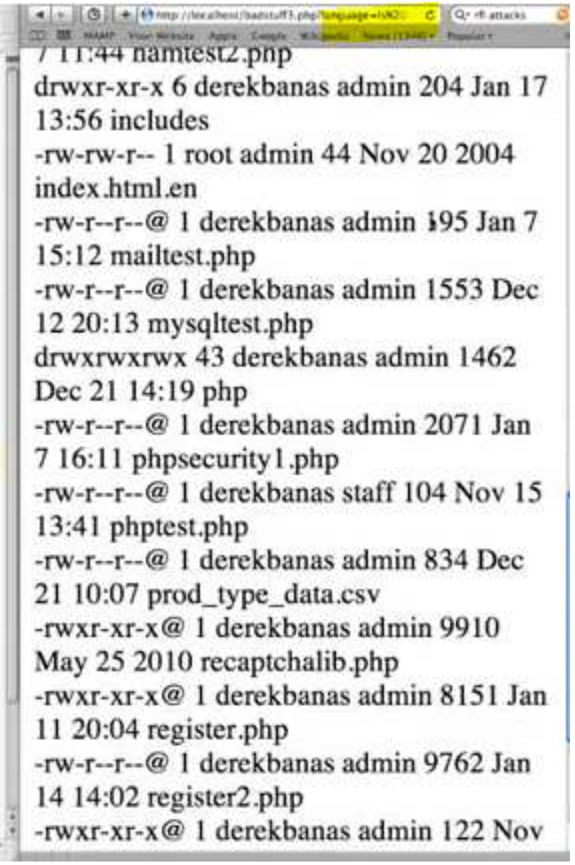
Comments

Bad Stuff

PHP SECU 18 _ This meant to show users data in ext files _ BUT language EQUAL ls -la _ list all finds in curr dir

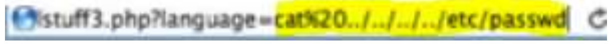


```
<html>
<head>
<title>Open for Monkey Business</title>
</head>
    <body>

    <?php
        if(isset($_GET['language']))
        {
            echo "<h2>Hacked</h2><br />";
            $language = $_GET['language'];
            exec($language, $execarray);
            foreach($execarray as $value)
            {
                echo $value . "<br />";
            }
        }

    ?>


    </body>
    </html>
```



```
7 11:44 namtest2.php
drwxr-xr-x 6 derekbanas admin 204 Jan 17
13:56 includes
-rw-rw-r-- 1 root admin 44 Nov 20 2004
index.html.en
-rw-r--r--@ 1 derekbanas admin 195 Jan 7
15:12 mailtest.php
-rw-r--r--@ 1 derekbanas admin 1553 Dec
12 20:13 mysqltest.php
drwxrwxrwx 43 derekbanas admin 1462
Dec 21 14:19 php
-rw-r--r--@ 1 derekbanas admin 2071 Jan
7 16:11 phpsecurity1.php
-rw-r--r--@ 1 derekbanas staff 104 Nov 15
13:41 phptest.php
-rw-r--r--@ 1 derekbanas admin 834 Dec
21 10:07 prod_type_data.csv
-rwxr-xr-x@ 1 derekbanas admin 9910
May 25 2010 recaptchalib.php
-rwxr-xr-x@ 1 derekbanas admin 8151 Jan
11 20:04 register.php
-rw-r--r--@ 1 derekbanas admin 9762 Jan
14 14:02 register2.php
-rwxr-xr-x@ 1 derekbanas admin 122 Nov
```

PHP SECU 18a _ hacker sees contents of users-log of the whole computer _ gotta regex out all those spec chars and their corres encoding



```
stuff3.php?language=cat%20../../../../etc/passwd
```

Stop Directory traversal

dont use funcs that interact with system

dont use shell cmds

no % < > / \ ; - ' & PLUS their encoding

websvr shud only have 1 owner named 'www', www should only own the web site folder. dont allow eg 'chown www /someOtherFolder' ie making www own someOtherFolder.

dont allow chmod functions. normal file priv is 755.

no: eval system exec file_get_contents passthru include include_once require require_once file_put_contents fopen

---

http://www.wikihow.com/Create-a-Secure-Login-Script-in-PHP-and-MySQL

1. **Newest versions** of PHP & SQL
2. **Database**:

- **Create DB:** Log into database as administrative user or root, `CREATE DATABASE `secure_login`;`

  Note: Some hosting services don't allow you to create a database through phpMyAdmin,Learn how to do it in cPanel.
- **Create a user:**
  - Use administrative user or root priviledges: **User:** "sec_user" **Password: "eKcGZr59zAa2BEWU"**
  - Create user that can't delete:

```
CREATE USER 'sec_user'@'localhost' IDENTIFIED BY 'eKcGZr59zAa2BEWU';
GRANT SELECT, INSERT, UPDATE ON `secure_login`.* TO 'sec_user'@'localhost';
```

- **Create table**

```
CREATE TABLE `secure_login`.`members` (
  `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `username` VARCHAR(30) NOT NULL,
  `email` VARCHAR(50) NOT NULL,
  `password` CHAR(128) NOT NULL,
  `salt` CHAR(128) NOT NULL
) ENGINE = InnoDB;
```

  - CHAR datatype: known length fields; saves on processing power
  - 128 characters long : as the fields "password" and "salt"
- **Create table for login attempts** to make brute force attacks more difficult:

```
CREATE TABLE `secure_login`.`login_attempts` (
  `user_id` INT(11) NOT NULL,
  `time` VARCHAR(30) NOT NULL
) ENGINE=InnoDB
```

- **Create test row to** test your login script, by creating a user with known details:

  The code you need in order to be able to log in as this user is:

```
INSERT INTO `secure_login`.`members` VALUES(1, 'test_user', 'test@example.com',
'00807432eae173f652f2064bdca1b61b290b52d40e429a7d295d76a71084aa96c0233b82f1feac45529e0726559645acaed6f3ae58a286b9f075916ebf66cacc',
'f9aab579fc1b41ed0c44fe4ecdbfcdb4cb99b9023abb241a6db833288f4eea3c02f76e0d35204a8695077dcf81932aa59006423976224be0390395bae152d4ef');
```

3. **Connection php file**

   psl-config.php: (put this outside of the web server's document root)

```
<?php
//These are the database login details
define("HOST", "localhost");     // The host you want to connect to.
define("USER", "sec_user");    // The database username.
define("PASSWORD", "4Fa98xkHVd2XmnfK");    // The database password.
define("DATABASE", "secure_login");    // The database name.
// global configuration variables. who can register, HTTPS or HTTP connection, database details etc
define("CAN_REGISTER", "any");
define("DEFAULT_ROLE", "member");
define("SECURE", FALSE);    // FOR DEVELOPMENT ONLY!!!!
?>
```

db_connect.php

```
<?php
include_once 'psl-config.php';   // As functions.php is not included
$mysqli = new mysqli(HOST, USER, PASSWORD, DATABASE);
```

4. Login functions: functions.php

- Better security and the privacy of your cookies:Create-a-Secure-Session-Managment-System-in-Php-and-Mysql.
- FLOW: Login form:
    - Have form data: sanitize the data.
    - Don’t have form data: Validate the registered user

- / : db_connect.php, register.php, login form, success,error
- /includes/ : psl-config.php, functions.php, process_login.php, logout.php, register.inc.php
- /js/ : sha512.js, forms.js

```
<?php
include_once 'psl-config.php';
//----SESSION---
function sec_session_start() { //NO “session_start()" to stop XSS (access session id in cookie using JS)
   $session_name = 'sec_session_id';   // Set a custom session name
   $secure = SECURE; //Use HTTPS too
   // This stops JavaScript being able to access the session id.
   $httponly = true;
   // Forces sessions to only use cookies.
   if (ini_set('session.use_only_cookies', 1) === FALSE) {
      header("Location: ../error.php?err=Could not initiate a safe session (ini_set)");
      exit();
   }
   // Gets current cookies params.
   $cookieParams = session_get_cookie_params();
   session_set_cookie_params($cookieParams["lifetime"],
      $cookieParams["path"],
      $cookieParams["domain"],
      $secure,
      $httponly);
   // Sets the session name to the one set above.
   session_name($session_name);
   session_start();           // Start the PHP session
   session_regenerate_id();    // New id upon reload, prevent session hijacking.
}
//----LOGIN---
function login($email, $password, $mysqli) {
   // Using prepared statements means that SQL injection is not possible.
   if ($stmt = $mysqli->prepare("SELECT id, username, password, salt
      FROM members
     WHERE email = ?
      LIMIT 1")) {
      $stmt->bind_param('s', $email);  // Bind "$email" to parameter.
      $stmt->execute();    // Execute the prepared query.
      $stmt->store_result();

      // get variables from result.
      $stmt->bind_result($user_id, $username, $db_password, $salt);
      $stmt->fetch();

      // hash the password with the unique salt.
      $password = hash('sha512', $password . $salt);
      if ($stmt->num_rows == 1) {
         // If the user exists we check if the account is locked
         // from too many login attempts

         if (checkbrute($user_id, $mysqli) == true) {
            // Account is locked
            // Send an email to user saying their account is locked
            return false;
         } else {
            // Check if the password in the database matches
            // the password the user submitted.
            if ($db_password == $password) {
               // Password is correct!
               // Get the user-agent string of the user.
               $user_browser = $_SERVER['HTTP_USER_AGENT'];
               // XSS protection as we might print this value
```

```
                $user_id = preg_replace("/[^0-9]+/", "", $user_id);
                $_SESSION['user_id'] = $user_id;
                // XSS protection as we might print this value
                $username = preg_replace("/[^a-zA-Z0-9_\-]+/",
                                                    "",
                                                    $username);
                $_SESSION['username'] = $username;
                $_SESSION['login_string'] = hash('sha512',
                       $password . $user_browser);
                // Login successful.
                return true;
             } else {
                // Password is not correct
                // We record this attempt in the database
                $now = time();
                $mysqli->query("INSERT INTO login_attempts(user_id, time)
                            VALUES ('$user_id', '$now')");
                return false;
             }
          }
       } else {
          // No user exists.
          return false;
       }
    }
}
/*   ----FOR BRUTE FORCE ATTACKS----
Log failed attempts and lock the user's account after five failed login attempts. This should trigger the sending of an email to the user with a reset link.
*/ALT: Display the CAPTCHA image after two failed login attempts. Use reCAPTCHA service from Google.
function checkbrute($user_id, $mysqli) {
    // Get timestamp of current time
    $now = time();

    // All login attempts are counted from the past 2 hours.
    $valid_attempts = $now - (2 * 60 * 60);

    if ($stmt = $mysqli->prepare("SELECT time
                      FROM login_attempts
```

```
                        WHERE user_id = ?
                       AND time > '$valid_attempts'")) {
        $stmt->bind_param('i', $user_id);

        // Execute the prepared query.
        $stmt->execute();
        $stmt->store_result();

        // If there have been more than 5 failed logins
        if ($stmt->num_rows > 5) {
            return true;
        } else {
            return false;
        }
    }
}
/*----CHECK LOGGED IN STATUS----
By checking "user_id" and the "login_string" SESSION variables. The "login_string" SESSION variable has the user's browser information hashed together with the
password. We use the browser information because it is very unlikely that the user will change their browser mid-session. Doing this helps prevent session
hijacking.
*/
function login_check($mysqli) {
    // Check if all session variables are set
    if (isset($_SESSION['user_id'],
                    $_SESSION['username'],
                    $_SESSION['login_string'])) {

        $user_id = $_SESSION['user_id'];
        $login_string = $_SESSION['login_string'];
        $username = $_SESSION['username'];

        // Get the user-agent string of the user.
        $user_browser = $_SERVER['HTTP_USER_AGENT'];

        if ($stmt = $mysqli->prepare("SELECT password
                                      FROM members
                                      WHERE id = ? LIMIT 1")) {
            // Bind "$user_id" to parameter.
            $stmt->bind_param('i', $user_id);
            $stmt->execute();   // Execute the prepared query.
            $stmt->store_result();

            if ($stmt->num_rows == 1) {
                // If the user exists get variables from result.
                $stmt->bind_result($password);
                $stmt->fetch();
                $login_check = hash('sha512', $password . $user_browser);

                if ($login_check == $login_string) {
                    // Logged In!!!!
                    return true;
                } else {
                    // Not logged in
                    return false;
                }
            } else {
                // Not logged in
                return false;
            }
        } else {
            // Not logged in
            return false;
        }
    } else {
        // Not logged in
        return false;
    }
}
/*----Sanitize URL from PHP_SELF (sanitizes the output from the PHP_SELF server variable)----
-The trouble with using the server variable unfiltered is that it can be used in a cross site scripting attack.
-Most references will simply tell you to filter it using htmlentities(), however even this appears not to be sufficient hence the belt and braces approach in this
function.

-Others suggest leaving the action attribute of the form blank, or set to a null string. Doing this, though, leaves the form open to an iframe clickjacking attack.
*/
function esc_url($url) {
    if ('' == $url) {
        return $url;
    }

    $url = preg_replace('|[^a-z0-9-~+_.?#=!&;,/:%@$\|*\'()\\x80-\\xff]|i', '', $url);

    $strip = array('%0d', '%0a', '%0D', '%0A');
    $url = (string) $url;

    $count = 1;
    while ($count) {
        $url = str_replace($strip, '', $url, $count);
    }

    $url = str_replace(';//', '://', $url);

    $url = htmlentities($url);

    $url = str_replace('&amp;', '&#038;', $url);
    $url = str_replace("'", '&#039;', $url);

    if ($url[0] !== '/') {
        // We're only interested in relative links from $_SERVER['PHP_SELF']
        return '';
    } else {
        return $url;
    }
}
```

5. Process_login.php

We will use the mysqli_* set of PHP functions as this is one of the most up-to-date mySQL extensions.

```
<?php
include_once 'db_connect.php';
include_once 'functions.php';

sec_session_start(); // Our custom secure way of starting a PHP session.

if (isset($_POST['email'], $_POST['p'])) {
    $email = $_POST['email'];
    $password = $_POST['p']; // The hashed password.

    if (login($email, $password, $mysqli) == true) {
        // Login success
        header('Location: ../protected_page.php');
    } else {
        // Login failed
        header('Location: ../index.php?error=1');
    }
} else {
    // The correct POST variables were not sent to this page.
    echo 'Invalid Request';
}
```

6. Logout script must start the session, destroy it and then redirect to somewhere else.
   Add CSRF protection here in case someone sends a link hidden in this page somehow. Coding Horror.
   logout.php:

```
<?php
include_once 'functions.php';
sec_session_start();

// Unset all session values
$_SESSION = array();

// get session parameters
$params = session_get_cookie_params();

// Delete the actual cookie.
setcookie(session_name(),
        '', time() - 42000,
        $params["path"],
        $params["domain"],
        $params["secure"],
        $params["httponly"]);

// Destroy session
session_destroy();
header('Location: ../index.php');
```

7. **Registration**: Use client side validation

register.php

```
<?php
include_once 'includes/register.inc.php';
include_once 'includes/functions.php';
?>
<!DOCTYPE html>
<html>
   <head>
      <meta charset="UTF-8">
      <title>Secure Login: Registration Form</title>
      <script type="text/JavaScript" src="js/sha512.js"></script>
      <script type="text/JavaScript" src="js/forms.js"></script>
      <link rel="stylesheet" href="styles/main.css" />
   </head>
   <body>
      <!-- Registration form to be output if the POST variables are not
      set or if the registration script caused an error. -->
      <h1>Register with us</h1>
      <?php
      if (!empty($error_msg)) {
         echo $error_msg;
      }
      ?>
      <ul>
         <li>Usernames may contain only digits, upper and lower case letters and underscores</li>
         <li>Emails must have a valid email format</li>
         <li>Passwords must be at least 6 characters long</li>
         <li>Passwords must contain
            <ul>
               <li>At least one upper case letter (A..Z)</li>
               <li>At least one lower case letter (a..z)</li>
               <li>At least one number (0..9)</li>
            </ul>
         </li>
         <li>Your password and confirmation must match exactly</li>
      </ul>
      <form action="<?php echo esc_url($_SERVER['PHP_SELF']); ?>"
```

```
            method="post"
            name="registration_form">
        Username: <input type='text'
            name='username'
            id='username' /><br>
        Email: <input type="text" name="email" id="email" /><br>
        Password: <input type="password"
                    name="password"
                    id="password"/><br>
        Confirm password: <input type="password"
                          name="confirmpwd"
                          id="confirmpwd" /><br>
        <input type="button"
              value="Register"
              onclick="return regformhash(this.form, //after submit for registration
                         this.form.username,
                         this.form.email,
                         this.form.password,
                         this.form.confirmpwd);" />
      </form>
      <p>Return to the <a href="index.php">login page</a>.</p>
   </body>
</html>
```

register.inc.php

```
<?php
include_once 'db_connect.php';
include_once 'psl-config.php';

$error_msg = "";

if (isset($_POST['username'], $_POST['email'], $_POST['p'])) {
   // Sanitize and validate the data passed in
   $username = filter_input(INPUT_POST, 'username', FILTER_SANITIZE_STRING);
   $email = filter_input(INPUT_POST, 'email', FILTER_SANITIZE_EMAIL);
   $email = filter_var($email, FILTER_VALIDATE_EMAIL);
   if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
      // Not a valid email
      $error_msg .= '<p class="error">The email address you entered is not valid</p>';
```

```
 }

 $password = filter_input(INPUT_POST, 'p', FILTER_SANITIZE_STRING);
 if (strlen($password) != 128) {
     // The hashed pwd should be 128 characters long.
     // If it's not, something really odd has happened
     $error_msg .= '<p class="error">Invalid password configuration.</p>';
 }

 // Username validity and password validity have been checked client side.
 // This should should be adequate as nobody gains any advantage from
 // breaking these rules.
 //

 $prep_stmt = "SELECT id FROM members WHERE email = ? LIMIT 1";
 $stmt = $mysqli->prepare($prep_stmt);

// check existing email
 if ($stmt) {
     $stmt->bind_param('s', $email);
     $stmt->execute();
     $stmt->store_result();

     if ($stmt->num_rows == 1) {
         // A user with this email address already exists
         $error_msg .= '<p class="error">A user with this email address already exists.</p>';
                 $stmt->close();
     }
             $stmt->close();
 } else {
     $error_msg .= '<p class="error">Database error Line 39</p>';
             $stmt->close();
 }

 // check existing username
 $prep_stmt = "SELECT id FROM members WHERE username = ? LIMIT 1";
 $stmt = $mysqli->prepare($prep_stmt);
```

```
if ($stmt) {
   $stmt->bind_param('s', $username);
   $stmt->execute();
   $stmt->store_result();

         if ($stmt->num_rows == 1) {
               // A user with this username already exists
               $error_msg .= '<p class="error">A user with this username already exists</p>';
               $stmt->close();
         }
         $stmt->close();
   } else {
         $error_msg .= '<p class="error">Database error line 55</p>';
         $stmt->close();
   }

// TODO:
// We'll also have to account for the situation where the user doesn't have
// rights to do registration, by checking what type of user is attempting to
// perform the operation.

if (empty($error_msg)) {
   // Create a random salt
   //$random_salt = hash('sha512', uniqid(openssl_random_pseudo_bytes(16), TRUE)); // Did not work
   $random_salt = hash('sha512', uniqid(mt_rand(1, mt_getrandmax()), true));

   // Create salted password
   $password = hash('sha512', $password . $random_salt);

   // Insert the new user into the database
   if ($insert_stmt = $mysqli->prepare("INSERT INTO members (username, email, password, salt) VALUES (?, ?, ?, ?)")) {
      $insert_stmt->bind_param('ssss', $username, $email, $password, $random_salt);
      // Execute the prepared query.
      if (! $insert_stmt->execute()) {
         header('Location: ../error.php?err=Registration failure: INSERT');
      }
   }
   header('Location: ./register_success.php');
```

```
    }
}
```

8. sha512.js
- This file is an implementation in JavaScript of the hashing algorithm sha512. We will use the hashing function so our passwords don't get sent in plain text.
- The file can be downloaded from pajhome.org.uk
- (It is also saved in the github repository)
- Store your copy of this file in a directory called "js", off the root directory of the application.

9. forms.js : hashes the passwords for the login (formhash()) and registration (regformhash()) forms & passes it in the POST data by creating and populating a hidden field.

```
function formhash(form, password) {
   // Create a new element input, this will be our hashed password field.
   var p = document.createElement("input");

   // Add the new element to our form.
   form.appendChild(p);
   p.name = "p";
   p.type = "hidden";
   p.value = hex_sha512(password.value);

   // Make sure the plaintext password doesn't get sent.
   password.value = "";

   // Finally submit the form.
   form.submit();
}

function regformhash(form, uid, email, password, conf) {
    // Check each field has a value
   if (uid.value == ''         ||
         email.value == ''     ||
         password.value == ''  ||
         conf.value == '') {

        alert('You must provide all the requested details. Please try again');
        return false;
   }

   // Check the username
```

```
re = /^\w+$/;
if(!re.test(form.username.value)) {
   alert("Username must contain only letters, numbers and underscores. Please try again");
   form.username.focus();
   return false;
}

// Check that the password is sufficiently long (min 6 chars)
// The check is duplicated below, but this is included to give more
// specific guidance to the user
if (password.value.length < 6) {
   alert('Passwords must be at least 6 characters long.  Please try again');
   form.password.focus();
   return false;
}

// At least one number, one lowercase and one uppercase letter
// At least six characters

var re = /(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{6,}/;
if (!re.test(password.value)) {
   alert('Passwords must contain at least one number, one lowercase and one uppercase letter.  Please try again');
   return false;
}

// Check password and confirmation are the same
if (password.value != conf.value) {
   alert('Your password and confirmation do not match. Please try again');
   form.password.focus();
   return false;
}

// Create a new element input, this will be our hashed password field.
var p = document.createElement("input");

// Add the new element to our form.
form.appendChild(p);
```

```
    p.name = "p";
    p.type = "hidden";
    p.value = hex_sha512(password.value);

    // Make sure the plaintext password doesn't get sent.
    password.value = "";
    conf.value = "";

    // Finally submit the form.
    form.submit();
    return true;
}
```

10. Login form:
    - When logging in, it is best to use something that is not public, for this guide we are using the email as the login id, the username can then be used to identify the user. If the email is not displayed on any pages within the wider application, it adds another unknown for anyone trying to crack the account.
    - Note: even though we have encrypted the password so it is not sent in plain text, it is essential that you use the HTTPS protocol (TLS/SSL) when sending passwords in a production system. It cannot be stressed enough that simply hashing the password is not enough. A man-in-the-middle attack could be mounted to read the hash being sent and use it to log in.

    Login.php

```
<?php
include_once 'includes/db_connect.php';
include_once 'includes/functions.php';

sec_session_start();

if (login_check($mysqli) == true) {
    $logged = 'in';
} else {
    $logged = 'out';
}
?>
<!DOCTYPE html>
<html>
    <head>
        <title>Secure Login: Log In</title>
        <link rel="stylesheet" href="styles/main.css" />
        <script type="text/JavaScript" src="js/sha512.js"></script>
        <script type="text/JavaScript" src="js/forms.js"></script>
    </head>
    <body>
        <?php
        if (isset($_GET['error'])) {
            echo '<p class="error">Error Logging In!</p>';
        }
        ?>
        <form action="includes/process_login.php" method="post" name="login_form">
            Email: <input type="text" name="email" />
            Password: <input type="password"
                             name="password"
                             id="password"/>
            <input type="button"
                   value="Login"
                   onclick="formhash(this.form, this.form.password);" />
        </form>
        <p>If you don't have a login, please <a href="register.php">register</a></p>
        <p>If you are done, please <a href="includes/logout.php">log out</a>.</p>
        <p>You are currently logged <?php echo $logged ?>.</p>
    </body>
</html>
```

Success or error php:

```
<?php
$error = filter_input(INPUT_GET, 'err', $filter = FILTER_SANITIZE_STRING);

if (! $error) {
    $error = 'Oops! An unknown error happened.';
}
?>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Secure Login: Error</title>
        <link rel="stylesheet" href="styles/main.css" />
    </head>
    <body>
        <h1>There was a problem</h1>
        <p class="error"><?php echo $error; ?></p>
    </body>
</html>
```

11. On every page after login,CHECK IF USER LOGGED IN!!!. EG:

```
<?php
include_once 'includes/db_connect.php';
include_once 'includes/functions.php';

sec_session_start();
?>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Secure Login: Protected Page</title>
        <link rel="stylesheet" href="styles/main.css" />
    </head>
    <body>
        <?php if (login_check($mysqli) == true) : ?>
            <p>Welcome <?php echo htmlentities($_SESSION['username']); ?>!</p>
            <p>
                This is an example protected page.  To access this page, users
                must be logged in.  At some stage, we'll also check the role of
                the user, so pages will be able to determine the type of user
                authorised to access the page.
            </p>
            <p>Return to <a href="index.php">login page</a></p>
        <?php else : ?>
            <p>
                <span class="error">You are not authorized to access this page.</span> Please <a href="index.php">login</a>.
            </p>
        <?php endif; ?>
    </body>
</html>
```

https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

This article is focused on providing clear, simple, actionable guidance for preventing SQL Injection flaws in your applications. SQL Injection attacks are unfortunately very common, and this is due to two factors:

1. the significant prevalence of SQL Injection vulnerabilities, and
2. the attractiveness of the target (i.e., the database typically contains all the interesting/critical data for your application).

It’s somewhat shameful that there are so many successful SQL Injection attacks occurring, because it is EXTREMELY simple to avoid SQL Injection vulnerabilities in your code.

SQL Injection flaws are introduced when software developers create dynamic database queries that include user supplied input. To avoid SQL injection flaws is simple. Developers need to either: a) stop writing dynamic queries; and/or b) prevent user supplied input which contains malicious SQL from affecting the logic of the executed query.

This article provides a set of simple techniques for preventing SQL Injection vulnerabilities by avoiding these two problems. These techniques can be used with practically any kind of programming language with any type of database. There are other types of databases, like XML databases, which can have similar problems (e.g., XPath and XQuery injection) and these techniques can be used to protect them as well.

Primary Defenses:

- **Option #1: Use of Prepared Statements (Parameterized Queries)**
- **Option #2: Use of Stored Procedures**
- **Option #3: Escaping all User Supplied Input**

Additional Defenses:

- **Also Enforce: Least Privilege**
- **Also Perform: White List Input Validation**

Unsafe Example

SQL injection flaws typically look like this:

The following (Java) example is UNSAFE, and would allow an attacker to inject code into the query that would be executed by the database. The unvalidated "customerName" parameter that is simply appended to the query allows an attacker to inject any SQL code they want. Unfortunately, this method for accessing databases is all too common.

```
String query = "SELECT account_balance FROM user_data WHERE user_name = "
  + request.getParameter("customerName");

try {
      Statement statement = connection.createStatement( … );
      ResultSet results = statement.executeQuery( query );
}
```

Primary Defenses

Defense Option 1: Prepared Statements (Parameterized Queries)

The use of prepared statements (aka parameterized queries) is how all developers should first be taught how to write database queries. They are simple to write, and easier to understand than dynamic queries. Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied.

Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker. In the safe example below, if an attacker were to enter the userID of tom' or '1'='1, the parameterized query would not be vulnerable and would instead look for a username which literally matched the entire string tom' or '1'='1.

Language specific recommendations:

- Java EE – use PreparedStatement() with bind variables
- .NET – use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
- PHP – use PDO with strongly typed parameterized queries (using bindParam())
- Hibernate - use createQuery() with bind variables (called named parameters in Hibernate)
- SQLite - use sqlite3_prepare() to create a statement object

In rare circumstances, prepared statements can harm performance. When confronted with this situation, it is best to either a) strongly validate all data or b) escape all user supplied input using an escaping routine specific to your database vendor as described below, rather than using a prepared statement. Another option which might solve your performance issue is to use a stored procedure instead.

Safe Java Prepared Statement Example

The following code example uses a PreparedStatement, Java's implementation of a parameterized query, to execute the same database query.

```
String custname = request.getParameter("customerName"); // This should REALLY be validated too
// perform input validation to detect attacks
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";

PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

Safe C# .NET Prepared Statement Example

With .NET, it's even more straightforward. The creation and execution of the query doesn't change. All you have to do is simply pass the parameters to the query using the Parameters.Add() call as shown here.

```
String query =
       "SELECT account_balance FROM user_data WHERE user_name = ?";
try {
      OleDbCommand command = new OleDbCommand(query, connection);
      command.Parameters.Add(new OleDbParameter("customerName", CustomerName Name.Text));
      OleDbDataReader reader = command.ExecuteReader();
      // …
} catch (OleDbException se) {
      // error handling
}
```

We have shown examples in Java and .NET but practically all other languages, including Cold Fusion, and Classic ASP, support parameterized query interfaces. Even SQL abstraction layers, like the Hibernate Query Language (HQL) have the same type of injection problems (which we call HQL Injection). HQL supports parameterized queries as well, so we can avoid this problem:

Hibernate Query Language (HQL) Prepared Statement (Named Parameters) Examples

```
First is an unsafe HQL Statement

Query unsafeHQLQuery = session.createQuery("from Inventory where productID='"+userSuppliedParameter+"'");

Here is a safe version of the same query using named parameters

Query safeHQLQuery = session.createQuery("from Inventory where productID=:productid");
safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

For examples of parameterized queries in other languages, including Ruby, PHP, Cold Fusion, and Perl, see the Query Parameterization Cheat Sheet.

Developers tend to like the Prepared Statement approach because all the SQL code stays within the application. This makes your application relatively database independent. However, other options allow you to store all the SQL code in the database itself, which has both security and non-security advantages. That approach, called Stored Procedures, is described next.

Defense Option 2: Stored Procedures

Stored procedures have the same effect as the use of prepared statements when implemented safely*. They require the developer to define the SQL code first, and then pass in the parameters after. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application. Both of these techniques have the same effectiveness in preventing SQL injection so your organization should choose which approach makes the most sense for you.

*Note: 'Implemented safely' means the stored procedure does not include any unsafe dynamic SQL generation. Developers do not usually generate dynamic SQL inside stored procedures. However, it can be done, but should be avoided. If it can't be avoided, the stored procedure must use input validation or proper escaping as described in this article to make sure that all user supplied input to the stored procedure can't be used to inject SQL code into the dynamically generated query. Auditors should always look for uses of sp_execute, execute or exec within SQL Server stored procedures. Similar audit guidelines are necessary for similar functions for other vendors.

There are also several cases where stored procedures can increase risk. For example, on MS SQL server, you have 3 main default roles: db_datareader, db_datawriter and db_owner. Before stored procedures came into use, DBA's would give db_datareader or db_datawriter rights to the webservice's user, depending on the requirements. However, stored procedures require execute rights, a role that is not available by default. Some setups where the user management has been centralized, but is limited to those 3 roles, cause all web apps to run under db_owner rights so stored procedures can work. Naturally, that means that if a server is breached the attacker has full rights to the database, where previously they might only have had read-access. More on this topic here. http://www.sqldbatips.com/showarticle.asp?ID=8

Safe Java Stored Procedure Example

The following code example uses a CallableStatement, Java's implementation of the stored procedure interface, to execute the same database query. The "sp_getAccountBalance" stored procedure would have to be predefined in the database and implement the same functionality as the query defined above.

```
String custname = request.getParameter("customerName"); // This should REALLY be validated
try {
      CallableStatement cs = connection.prepareCall("{call sp_getAccountBalance(?)}");
      cs.setString(1, custname);
      ResultSet results = cs.executeQuery();
      // … result set handling
} catch (SQLException se) {
      // … logging and error handling
}
```

Safe VB .NET Stored Procedure Example

The following code example uses a SqlCommand, .NET’s implementation of the stored procedure interface, to execute the same database query. The "sp_getAccountBalance" stored procedure would have to be predefined in the database and implement the same functionality as the query defined above.

```
Try
      Dim command As SqlCommand = new SqlCommand("sp_getAccountBalance", connection)
      command.CommandType = CommandType.StoredProcedure
      command.Parameters.Add(new SqlParameter("@CustomerName", CustomerName.Text))
      Dim reader As SqlDataReader = command.ExecuteReader()
      ‘ …
Catch se As SqlException
```

```
        ` error handling
End Try
```

We have shown examples in Java and .NET but practically all other languages, including Cold Fusion, and Classic ASP, support the ability to invoke stored procedures.

For organizations that already make significant or even exclusive use of stored procedures, it is far less likely that they have SQL injection flaws in the first place. However, you still need to be careful with stored procedures because it is possible, although relatively rare, to **create a dynamic query inside of a stored procedure that is subject to SQL injection**. If dynamic queries in your stored procedures can't be avoided, you can use bind variables inside your stored procedures, just like in a prepared statement. Alternatively, you can validate or properly escape all user supplied input to the dynamic query, before you construct it. For examples of the use of bind variables inside of a stored procedure, see the Stored Procedure Examples in the OWASP Query Parameterization Cheat Sheet.

There are also some additional security and non-security benefits of stored procedures that are worth considering. One security benefit is that if you make exclusive use of stored procedures for your database, you can restrict all database user accounts to only have access to the stored procedures. This means that database accounts do not have permission to submit dynamic queries to the database, giving you far greater confidence that you do not have any SQL injection vulnerabilities in the applications that access that database. Some non-security benefits include performance benefits (in most situations), and having all the SQL code in one location, potentially simplifying maintenance of the code and keeping the SQL code out of the application developers' hands, leaving it for the database developers to develop and maintain.

Defense Option 3: Escaping All User Supplied Input

This third technique is to escape user input before putting it in a query. If you are concerned that rewriting your dynamic queries as prepared statements or stored procedures might break your application or adversely affect performance, then this might be the best approach for you. However, this methodology is frail compared to using parameterized queries and we cannot guarantee it will prevent all SQL Injection in all situations. This technique should only be used, with caution, to retrofit legacy code in a cost effective way. Applications built from scratch, or applications requiring low risk tolerance should be built or re-written using parameterized queries.

This technique works like this. Each DBMS supports one or more character escaping schemes specific to certain kinds of queries. If you then escape all user supplied input using the proper escaping scheme for the database you are using, the DBMS will not confuse that input with SQL code written by the developer, thus avoiding any possible SQL injection vulnerabilities.

- Full details on ESAPI are available here on OWASP.

- The javadoc for ESAPI is available here at its Google Code repository.
- You can also directly browse the source at Google, which is frequently helpful if the javadoc isn't perfectly clear.

To find the javadoc specifically for the database encoders, click on the ‘Codec’ class on the left hand side. There are lots of Codecs implemented. The two Database specific codecs are OracleCodec, and MySQLCodec.

Just click on their names in the ‘All Known Implementing Classes:’ at the top of the Interface Codec page.

At this time, ESAPI currently has database encoders for:

- Oracle
- MySQL (Both ANSI and native modes are supported)

Database encoders for:

- SQL Server
- PostgreSQL

Are forthcoming. If your database encoder is missing, please let us know.

Database Specific Escaping Details

If you want to build your own escaping routines, here are the escaping details for each of the databases that we have developed ESAPI Encoders for:

Oracle Escaping

This information is based on the Oracle Escape character information found here:http://www.orafaq.com/wiki/SQL_FAQ#How_does_one_escape_special_characters_when_writing_SQL_queries.3F

Escaping Dynamic Queries

To use an ESAPI database codec is pretty simple. An Oracle example looks something like:

```
ESAPI.encoder().encodeForSQL( new OracleCodec(), queryparam );
```

So, if you had an existing Dynamic query being generated in your code that was going to Oracle that looked like this:

```
String query = "SELECT user_id FROM user_data WHERE user_name = '" + req.getParameter("userID")
+ "' and user_password = '" + req.getParameter("pwd") +"'";
try {
    Statement statement = connection.createStatement( … );
    ResultSet results = statement.executeQuery( query );
}
```

You would rewrite the first line to look like this:

```
Codec ORACLE_CODEC = new OracleCodec();
 String query = "SELECT user_id FROM user_data WHERE user_name = '" +
   ESAPI.encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("userID")) + "' and user_password = '"
   + ESAPI.encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("pwd")) +"'";
```

And it would now be safe from SQL injection, regardless of the input supplied.

For maximum code readability, you could also construct your own OracleEncoder.

```
Encoder oe = new OracleEncoder();
String query = "SELECT user_id FROM user_data WHERE user_name = '"
  + oe.encode( req.getParameter("userID")) + "' and user_password = '"
  + oe.encode( req.getParameter("pwd")) +"'";
```

With this type of solution, all your developers would have to do is wrap each user supplied parameter being passed in into an **ESAPI.encoder().encodeForOracle( )** call or whatever you named it, and you would be done.

Turn off character replacement

Use SET DEFINE OFF or SET SCAN OFF to ensure that automatic character replacement is turned off. If this character replacement is turned on, the & character will be treated like a SQLPlus variable prefix that could allow an attacker to retrieve private data.

See http://download.oracle.com/docs/cd/B19306_01/server.102/b14357/ch12040.htm#i2698854 and http://stackoverflow.com/questions/152837/how-to-insert-a-string-which-contains-an for more information

Escaping Wildcard characters in Like Clauses

The LIKE keyword allows for text scanning searches. In Oracle, the underscore '_' character matches only one character, while the ampersand '%' is used to match zero or more occurrences of any characters. These characters must be escaped in LIKE clause criteria. For example:

```
SELECT name FROM emp
WHERE id LIKE '%/_%' ESCAPE '/';
SELECT name FROM emp
WHERE id LIKE '%\%%' ESCAPE '\';
```

Oracle 10g escaping

An alternative for Oracle 10g and later is to place { and } around the string to escape the entire string. However, you have to be careful that there isn't a } character already in the string. You must search for these and if there is one, then you must replace it with }}. Otherwise that character will end the escaping early, and may introduce a vulnerability.

MySQL Escaping

MySQL supports two escaping modes:

1. ANSI_QUOTES SQL mode, and a mode with this off, which we call
2. MySQL mode.

ANSI SQL mode: Simply encode all ' (single tick) characters with '' (two single ticks)

MySQL mode, do the following:

```
 NUL (0x00) --> \0  [This is a zero, not the letter O]
 BS  (0x08) --> \b
 TAB (0x09) --> \t
 LF  (0x0a) --> \n
 CR  (0x0d) --> \r
 SUB (0x1a) --> \Z
 "   (0x22) --> \"
 %   (0x25) --> \%
```

```
'   (0x27) --> \'
\   (0x5c) --> \\
_   (0x5f) --> \_
all other non-alphanumeric characters with ASCII values less than 256  --> \c
where 'c' is the original non-alphanumeric character.
```

This information is based on the MySQL Escape character information found here: http://mirror.yandex.ru/mirrors/ftp.mysql.com/doc/refman/5.0/en/string-syntax.html

SQL Server Escaping

We have not implemented the SQL Server escaping routine yet, but the following has good pointers to articles describing how to prevent SQL injection attacks on SQL server

- http://blogs.msdn.com/raulga/archive/2007/01/04/dynamic-sql-sql-injection.aspx

DB2 Escaping

This information is based on DB2 WebQuery special characters found here: https://www-304.ibm.com/support/docview.wss?uid=nas14488c61e3223e8a78625744f00782983 as well as some information from Oracle's JDBC DB2 driver found here: http://docs.oracle.com/cd/E12840_01/wls/docs103/jdbc_drivers/sqlescape.html

Information in regards to differences between several DB2 Universal drivers can be found here: http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/ad/rjvjcsqc.htm

Additional Defenses

Beyond adopting one of the three primary defenses, we also recommend adopting all of these additional defenses in order to provide defense in depth. These additional defenses are:

- **Least Privilege**
- **White List Input Validation**

Least Privilege

To minimize the potential damage of a successful SQL injection attack, you should minimize the privileges assigned to every database account in your environment. Do not assign DBA or admin type access rights to your application accounts. We understand that this is easy, and everything just ‘works’ when you do it this way, but it is very dangerous. Start from the ground up to determine what access rights your application accounts require, rather than trying to figure out what access rights you need to take away. Make sure that accounts that only need read access are only granted read access to the tables they need access to. If an account only needs access to portions of a table, consider creating a view that limits access to that portion of the data and assigning the account access to the view instead, rather than the underlying table. Rarely, if ever, grant create or delete access to database accounts.

If you adopt a policy where you use stored procedures everywhere, and don’t allow application accounts to directly execute their own queries, then restrict those accounts to only be able to execute the stored procedures they need. Don’t grant them any rights directly to the tables in the database.

SQL injection is not the only threat to your database data. Attackers can simply change the parameter values from one of the legal values they are presented with, to a value that is unauthorized for them, but the application itself might be authorized to access. As such, minimizing the privileges granted to your application will reduce the likelihood of such unauthorized access attempts, even when an attacker is not trying to use SQL injection as part of their exploit.

While you are at it, you should minimize the privileges of the operating system account that the DBMS runs under. Don't run your DBMS as root or system! Most DBMSs run out of the box with a very powerful system account. For example, MySQL runs as system on Windows by default! Change the DBMS's OS account to something more appropriate, with restricted privileges.

White List Input Validation

Input validation can be used to detect unauthorized input before it is passed to the SQL query. For more information please see the Input Validation Cheat Sheet.

Related Articles

SQL Injection Attack Cheat Sheets

The following articles describe how to exploit different kinds of SQL Injection Vulnerabilities on various platforms that this article was created to help you avoid:

- Ferruh Mavituna : "SQL Injection Cheat Sheet" - http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- RSnake : "SQL Injection Cheat Sheet-Esp: for filter evasion" - http://ha.ckers.org/sqlinjection/

Description of SQL Injection Vulnerabilities

- OWASP article on SQL Injection Vulnerabilities
- OWASP article on Blind_SQL_Injection Vulnerabilities

How to Avoid SQL Injection Vulnerabilities

- OWASP Developers Guide article on how to Avoid SQL Injection Vulnerabilities
- OWASP article on Preventing SQL Injection in Java
- OWASP Cheat Sheet that provides numerous language specific examples of parameterized queries using both Prepared Statements and Stored Procedures
- The Bobby Tables site (inspired by the XKCD webcomic) has numerous examples in different languages of parameterized Prepared Statements and Stored Procedures

How to Review Code for SQL Injection Vulnerabilities

- OWASP Code Review Guide article on how to Review Code for SQL Injection Vulnerabilities

How to Test for SQL Injection Vulnerabilities

- OWASP Testing Guide article on how to Test for SQL Injection Vulnerabilities