**SECURITY METHODS**

**Preventing spam**

1. Validate any email addresses using regular expressions or the filter extension.
2. If a value contains anything from the list below, don’t use that value in a sent email:

|  |  |  |
| --- | --- | --- |
| ‘to:’ | ‘cc:’ | ‘bcc:’ |
| ‘content-type:’ | ‘mime-version:’ | ‘multipart-mixed:’ |
| ‘content-transfer-encoding:’ | \r  \n  %0a  %0d  (all used to create new lines) |  |

Example

If($\_SERVER[‘request\_method’] == “POST”) {

Function spam\_scrubber($value) {

//list of very bad values

$very\_bad = array(‘to:’, ’cc:’, ’bcc:’, ‘content-type:’, ‘mime-version:’, ‘multipart- mixed:’, ‘content-transfer-encoding:’);

//if any of the very bad strings are in the submitted value, return an empty string

Foreach ($very\_bad as $v) {

If(stripos($value, $v) !== false) return ‘’;

} //stripos performs a case-insensitive search (e.g. bcc, BcC etc… all the same)

//replace any newline characters with spaces:

//str\_replace() looks through the value in the 3rd argument and replaces any occurences of the characters in the 1st argument with the character of characters in the 2nd.

$value = str\_replace(array(“\r”, “\n”, “%0a”, “%0d”), ‘ ‘, $value);

//return the value

Return trim($value);

} // end of spam\_Scrubber function

//clean the form data

//array\_map() has 2 arguments: 1 – the name of the function to call, 2 -an array

$scrubbed = array\_map(‘spam\_scrubber’, $\_POST);

//minimal form validation

If(!empty($scrubbed[‘name’]) && !empty($scrubbed[‘email’]) && !empty($scrubbed[‘comments’]) ) {

//create the body

$body = “Name: {$scrubbed[‘name’]}\n\nComments: {$scrubbed[‘comments’]}”;

//make it no longer than 70 characters long

$body = wordwrap($body, 70);

//send the email

Mail(‘your\_email@example.com’, ’Contact form submission’, $body, “from: {$scrubbed[‘email’]}”);

//print message

Echo “Thank your for contacting me. I will reply some day”;

//clear post (so that the forms not sticky)

$\_POST = array();

}else{

Echo “Please fill out the form completely”;

}

}//end of main isset() IF

**Validating data by type**

|  |  |
| --- | --- |
| **Function** | **Checks for** |
| Is\_array() | Arrays |
| Is\_bool() | Booleans |
| Is\_float() | Floating point numbers |
| Is\_int() | Integers |
| Is\_null() | Nulls |
| Is\_numeric() | Numeric values |
| Is\_resource() | Resources (like a DB connection) |
| Is\_scalar() | Scalar (single-valued) variables |
| is\_string() | strings |

The 2 types of validation approaches:

1. Blacklist
2. Whitelist

1 - these values are bad, anything else is good

2 – these values are good, anything else is bad

In PHP you can even change a variables type, after its been assigned a value. This is called typecasting.

$var == 20.2;

Echo (int) $var; //20

$quantity = (int) $\_POST[‘quanitity’];

\*You should definitely useteting when working with numbers within SQL queries\*

**Validating files by type**

Uploading files allows users to place a more potent type of content on your server (compared with just the text sent via a form), one cannot be too mindful of security when it comes to handling them.

The fileinfo extension determines a files type (and encoding) by hunting for ‘magic bytes’ or ‘magic numbers’ within the file. E.g. the data that makes up a GIF image must begin with the ASCII code that represents either GIF89a or GIF87a, the data that makes up a PDF file starts with %PDF.

To use fileinfo:

1. Create a fileinfo resource. The *kind* value will be one of several constants, indicating the type of resource you want to create. To determine a files type, the constant is FILEINFO\_MIME\_TYPE.

$fileinfo = finfo\_open(*kind*);

$fileinfo = finfo\_open(FILEINFO\_MIME\_TYPE);

1. Call the finfo\_file() function, providing the fileinfo resource & a reference to the file you want to examine:

Finfo\_file($fileinfo, $filename);

1. Close the fileinfo resource:

Finfo\_close($fileinfo);

\*The fileinfo DLL file should be included with your installation, but you may need to enable it in your php.ini configuration file\*

Example (confirm that an uploaded file is an RTF (rich text format) – p415-417(full example)

$fileinfo = finfo\_open(FILEINFO\_MIME\_TYPE);

If(finfo\_file($fileinfo, $\_FILES[‘upload’][‘tmp\_name’]) == ‘text/rtf’) {

//its ok

}else{

//inform user wrong format

}

Finfo\_close($fileinfo);

**Preventing XSS attacks**

Many dynamically driven web applications take the information submitted by a user, store it in a database, and then redisplay that information on another page. Think of a forum, a user could enter HTML code in their data/input. Javascript is also just plain text – if malicious code entered into a form were redisplayed in a web-browser, it could create pop up windows, steal cookies or redirect the browser to other sites. These attacks are referred to as cross-site scripting (XSS).

PHP includes a handful of functions for handling HTML and other code found within strings:

Htmlspecialchars() , which turns &, ‘, “, < and > into HTML entity format (&amp;, $quot;, etc…)

Htmlentities() , which turns all applicable characters into their HTML entity format

Strip\_tags() , which removes all HTML and PHP tags

The strip\_tags() function takes an optional parameter indicating what tags should not be stripped:

$var = strip\_tags($var, ‘<p><br/>’);

Example (p419-420)

Htmlentities($\_POST[‘data’]);

Strip\_tags($\_POST[‘data’]);

**Using the filter extension**

An important tool which bridges the gap between the relatively simple approach of typecasting and the more complex concept of regular expressions.

The filter extension can be used for one of 2 purposes: validating data, or sanitising it.

Sanitising data refers to altering data by removing inappropriate characters in order to make the data meet expectations.

|  |
| --- |
| **Validation filters** |
| FILTER\_VALIDATE\_BOOLEAN |
| FILTER\_VALIDATE\_EMAIL |
| FILTER\_VALIDATE\_FLOAT |
| FILTER\_VALIDATE\_INT |
| FILTER\_VALIDATE\_IP |
| FILTER\_VALIDATE\_REGEXP |
| FILTER\_VALIDATE\_URL |

|  |
| --- |
| **Sanitisation filters** |
| FILTER\_SANITIZE\_EMAIL |
| FILTER\_SANITIZE\_ENCODED |
| FILTER\_SANITIZE\_MAGIC\_QUOTES |
| FILTER\_SANITIZE\_NUMBER\_FLOAT |
| FILTER\_SANITIZE\_NUMBER\_INT |
| FILTER\_SANITIZE\_SPECIAL\_CHARS |
| FILTER\_SANITIZE\_STRING |
| FILTER\_SANITIZE\_STRIPPED |
| FILTER\_SANITIZE\_URL |
| FILTER\_UNSAFE\_RAW |

The most important function in the filter extension is filter\_var():

Filter\_var(var, filter[,options]);

Where var = variable to be filtered

Where filter = filter to apply

Where options = additional criteria

E.g. to confirm that a variable has a decimal value:

If(filter\_var($var, FILTER\_VALIDATE\_FLOAT)) {

//code

}

\*see examples on p421-424 for more information\*

**Preventing SQL injection attacks**

*Endeavours to insert bad code into a sites SQL queries.*

Fortunately, SQL injection attacks are rather easy to prevent:

1. Validate all data to be used in queries (& perform typecasting or perform the filter extension, whenever possible)
2. Use mysqli\_real\_escape\_string() function which makes data safe to use in queries
3. Don’t show detailed errors on live sites

Prepared statement performance

Prepared statement performance can be more secure than running queries in the old-fashioned way, but they may also be faster. If a PHP script sends the same query to MySQL multiple times, using different values each time, prepared statements can really speed things up. In such cases, the query itself is only sent to MySQL and parsed once. Then, the values are sent to MySQL separately.

e.g. the following code would run 100 queries in MySQL:

$q = “INSERT INTO counter(num) VALUES (?)”;

$stmt = mysqli\_prepare($dbc, $q);

Mysqli\_stmt\_bind\_param($stmt, ‘i’, $n);

For($n = 1; $n <=100; $n++) {

Mysqli\_stmt\_execute($stmt);

}

Even though the query is being run 100 times, the full text is only being transferred to, and parsed by, MySQL once. MySQL versions 5.1.17 and later include a caching mechanism that may also improve the performance of other uses of prepared statements.

Preventing brute force attacks

A brute force attack is an attempt to log in to a secure system by making lots of attempts in the hopes of eventual success. The best way to prevent brute force attacks from succeeding is requiring users to register with good, hard-to-guess passwords: containing letters, numbers, and punctuation; both upper and lower case; words not in the dictionary; at least 8 characters long, etc. Also, don’t give indication as to why a login failed: saying that a username and password combination isn’t correct gives away nothing, but saying that a username isn’t right or that the password isn’t right for that username says too much.

You could also limit the number of incorrect login attempts by a given IP address.

**Test**

1. Go through preventing spam example above until understood
2. List the functions used to validate data by type and give brief examples
3. Give example of typecasting
4. Give example of how you would only be able to upload files of a certain format
5. What are XSS attacks and what 3 functions can be used to fight against it. Give examples
6. Give an example of filter extension using the table above
7. What are SQL injection attacks and what 3 things can you do to prevent against them?
8. What is prepared statement performance, and give brief example
9. How can you prevent brute force attacks?