PHP II

Objectives

We've seen:

- Basic PHP syntax
- Basic Authentication with PHP
  - login and logout

Coming up:

- User registration and data validation
- PHP Slideshow
- File IO
- More advanced authentication: Hashing passwords

An aside

Japanese Website Design

Basic Authorization

Login page:

- start a session
- have the user enter a username and password
- check them in php
  - if valid
    - set session variables
    - proceed to login-only page
  - if invalid, return to login page

Basic Authorization

In pages requiring login

- start the session
- check that the session variables are set and match current user
- if valid: load the page
- if not valid, return to the login page

Basic Authorization

In logout page

- start the session
- unset session variables
- destroy the session
- return to login page (or somewhere else)

File IO with PHP

We often need to access files

PHP provides ways to:

- test files (file_exists, is_dir, ...)
- manipulate files (copy, unlink (deletes a file), chmod, ...)
- File IO (fopen, fclose, feof (tests end of file), ...)
- directory handling (mkdir, rmdir, opendir, closedir, ...)
- file status (fileatime (last access time), filemtime (file modification time)

Example: PHP built slideshow

First, look at how’d we do it without file access server side…

PHP Slideshow demo

PHP generating html

We have seen how to use PHP to generate html

We can also use loops and conditionals to generate content

Demo

- use data entered by user in a form to generate content
- LoopsETC.php

File manipulation

Filemanip.php

A first step towards a better login

Previous Login example stored login and password in plain text

It is better to encrypt passwords

Store them in a separate file

Passwords

Many sites have some form of user validation

Common to use usernames and passwords

For sites using databases, it makes sense to store this data in the database (we'll get to this in a bit)

But!!

- Never ever ever store plain text passwords in your database
- Need to encrypt them

Encrypting Passwords

It is typical to use 1-way encryption for passwords

Not even we can decrypt them, instead, we encrypt password attempts and compare the stored password and the new one

We store the encrypted password

How to choose a hash algorithm

There are many different hashing algorithms

- all have different purposes and meet different needs

Not all hashing algorithms are good for passwords

Good candidates:

- md5 (not recommended anymore)
- sha-1
- sha-2 (sha-256, sha-512)
- whirlpool, Tiger, AES
- Blowfish

Blowfish

Advantages:

- built in to php (so are others)
- high level of security
- public domain
- no patents
- free to use
- slow

More info:

- https://en.wikipedia.org/wiki/Bcrypt
- https://en.wikipedia.org/wiki/Blowfish_(cipher)

Encryption Algorithms in PHP

Many built in:

- md5($password);
- sha1($password);
- hash('sha1', $password);
  - http://php.net/manual/en/function.hash.php
  - hash_algos() will give you more info about available hash algorithms
    - http://php.net/manual/en/function.hash-algos.php
  - very fast
  - not suitable for passwords
- crypt($password, $salt);
  - slow (which is good)

crypt function

supports 6 different encryption algorithms:

- DES, Ext-DES, MD5, SHA-256, SHA-512, Blowfish
- Note that these algorithms will work differently when using crypt than when using hash()
  - may be run multiple times making them slower and changing the output
- We don’t have to pass in to crypt which algorithm we want (directly)
  - it gets included in the salt

Salting passwords

A little aside about Rainbow tables...

Rainbow Tables

Suppose someone (a bad man) gets access to our hashed passwords

They could take an entry (a hashed password) and try all possible passwords, hashing them until they find a hash that matches

- This kills the password (reveals it)
- This is slow, but...
  - we could build a table ahead of time: Rainbow Table
  - we then just have to look for the hash (suddenly, a linear time solution appears!)

Salts help us prevent this kind of attack

Salts

A salt is just some extra data that is “added” to the password before it is encrypted

Renders rainbow tables less appealing

For example we could turn $password into:

- “Put salt on my {$password}”

Unique salts

Another option is to make a salt uniquely for each user

- “put salt on {$password} for {$user}”
- this renders rainbow tables even less useful, because a separate table for each salt (or a way way way bigger table)

We can also add pseudo-random strings to salts

- “put salt on {$password} for {$user} at” . time();
- discovering the password requires knowing the random string
- but how do we figure out the salt later?
  - commonly store the salt as well
  - can be stored with the encrypted password
  - can also hash the salt!

Let’s try it

First demo we’ll play with passwords, hashing and creating salts

Second demo, we’ll generalize and create reusable function

- which should be stored somewhere secure

Salts

Why is it ok to store the salt in plaintext with the password?