LOGIN / REGISTRATION SCRIPT TUTORIAL

*Contributed by DevNet members **Celauran** and **social_experiment***
2012

Creating login systems can be a challenge for people who are new to PHP. For their benefit we decided to create an article showing some points to look at when creating a login system. These are common pitfalls identified by looking at previous questions from newbies who attempted to write login systems but got stuck somewhere along the way. This may cause some to continue using poorly crafted scripts, only to realize later (possibly after a cracking attempt) that the script wasn't properly secured.

Because this is aimed at less experienced users, a few things to point out before starting:

1. Any page that uses sessions has to have the **session_start()** function at the top (before any other output except the php opening tag) of that specific page.

```
<?php
 session_start();

 // rest of code
?>
```

If you have 5 pages to protect, all 5 pages needs session_start() at the top.

2. When using **mysql_** or **mysqli_** functions, you need to make a connection to the database. Certain **mysqli_** functions require the use of a resource returned by **mysqli_connect()** [http://php.net/mysqli_connect]. Throughout this article the presence of a database connection is assumed.

Pitfall 1: Storing passwords

The short answer is don't. You don't store passwords anywhere. Ever. Why? If your database is ever compromised, then every user account is also compromised. A list of usernames, passwords, and possibly email addresses in the hands of an attacker can be disastrous. Still, for a user to access your site, you need to validate the credentials they provide at login against those they provided at registration. The solution, then, is to use hashes. Store a hash of the password in your database, hash the password provided at login, and compare the hashes. Because hashing algorithms are one-way, computing the hash of any given value is trivial, but working backwards from the hash to the original value is impossible.

Still, not all hashing algorithms are created equal. For example, many tutorials -- and indeed some published works -- advocate the use of the md5 hashing algorithm for creating password hashes. While this may once have been acceptable practice, this is no longer the case. More on that later. For now, to create a reasonably secure working hash, we'll create a salt, create a pepper, combine these with the user's password, and finally hash with something like the sha384 algorithm (minimum). Salts are user-specific and can be stored in the database, while peppers are site-wide and often reside in a file somewhere or as a string which has various characters and is at least 30 characters long.

One key point to remember here is that you aren't trying to protect against unauthorized logins to your site; you're trying to prevent passwords from being discovered in the event that your user database is compromised. To that end, you want to employ a slow hashing algorithm. [url=http://codahale.com/how-

to-safely-store-a-password/]Read why[/url]. Take 5 minutes to read that article. Go ahead, I'll wait. You can use bcrypt directly by calling the [url=http://php.net/crypt]crypt()[/url] function with a suitable salt, or you can take advantage of [url=http://www.openwall.com/phpass]PHPass[/url] which uses bcrypt when available and degrades gracefully where it isn't. They also have [url=http://www.openwall.com/articles/PHP-Users-Passwords]an excellent article[/url] on hashing and user authentication.

Example of how you can create a hashed value:

```
<?php
 $hashedValue = hash('sha384', $salt.$pepper.$password);
 // returns a 96 character string which is stored in the database
?>
```

Alternately, using PHPass:
[syntax=php]$hasher = new PasswordHash(8, FALSE);
$hash   = $hasher->HashPassword($password);[/syntax]

It's worth noting that any given algorithm will produce hashes of the same length regardless of the length of the input. These will often be considerably longer than the length of the password being hashed. Be sure to check the length of the output of your algorithm of choice and ensure that the password field in your database is sufficiently long to store the entire hash.

Pitfall 2: Not escaping input

Quite a few example scripts portray a query against the database in the following manner:

```
<?php
$username = $_POST['username'];
$email    = $_POST['email'];
$qry = "SELECT * FROM Table WHERE username='$username' and email='$email'";
?>
```

The $_POST values above are taken directly from the form, without any checking of any sort. Input should always be treated as if it is contaminated. Before using data in a database query, it needs to be validated and escaped.

You can use mysql_real_escape_string() [http://php.net/mysql_real_escape_string] for this purpose. MySQL**i** [http://php.net/mysqli_real_escape_string] (which you really ought to be using) has a similar function. Better still, make use of prepared statements. Regardless of which method you ultimately choose, the value of properly escaping your data cannot be overstated. [http://xkcd.com/327/].

mysql_real_escape_string() accepts an argument that is to be escaped making it safe to use in a database query. The explanation from the PHP Manual on the function:
“Escapes special characters in the unescaped_string , taking into account the current character set of the connection so that it is safe to place it in a mysql_query(). If binary data is to be inserted, this function must be used. mysql_real_escape_string() calls MySQL's library function mysql_real_escape_string, which prepends backslashes to the following characters: \x00, \n, \r, \, ', " and \x1a. This function must always (with few exceptions) be used to make data safe before sending a query to MySQL.”

Depending on personal requirements you should also check for empty fields, certain types of characters, certain types of data. Checking the data you receive is just as important as escaping it. Use existing php functions such as trim() [http://php.net/trim], ctype functions [http://php.net/manual/en/book.ctype.php], filters[http://php.net/filter_var], or regular expressions to ensure that input is valid and of the type expected. User input is NEVER to be trusted.

Some scripts rely on the magic_quotes_gpc setting to determine whether or not to use mysql_real_escape_string(); though you could check for the existence of the value, it is wise to note that from PHP 5.3.0 the feature is deprecated (the function shouldn't be used anymore). The above code snippet can be amended as follows:

```
<?php
// no checking of data; improve this by using existing or custom
// functions.
$username = mysql_real_escape_string($_POST['username']);
$email    = mysql_real_escape_string($_POST['email']);
$qry = "SELECT columnA, columnB FROM TableName WHERE username='$username'
     and email='$email'";
?>
```

Pitfall 3: Session vulnerabilities

Sessions are commonly used to distinguish authenticated users from the unauthenticated. Typically, in processing a login form, you'll see something like this

```
<?php
if ($rows==1)
{
    header("location:/login_success.php");
}
else
{
    echo "Wrong Username or Password";
}
?>
```

Unfortunately, this leaves you vulnerable to *session fixation attacks* [http://www.acros.si/papers/session_fixation.pdf]. As captured session IDs are generally worthless unless the user is signed in, you can protect against this by regenerating the session ID upon successful login. Session data is only written after this new ID has been generated.

```
<?php
if($rows==1)
{
    session_regenerate_id();
    $_SESSION['user_id']  = $user_id;
    $_SESSION['loggedIn'] = true;
    // close the session
    session_write_close();
```

```
    header("location:/login_success.php");
    exit();
}
else
{
    echo "Wrong Username or Password";
}
?>
```

To additionally protect against session hijacking, add some sort of signature to the session. A combination of user ID, User-Agent, and some random salt should suffice. So we update the above to include this signature.

```
<?php
if($rows==1)
{
    session_regenerate_id();
    $_SESSION['user_id']   = $user_id;
    $_SESSION['loggedIn']  = true;
    $_SESSION['signature'] = md5($user_id . $_SERVER['HTTP_USER_AGENT'] .
$salt);
    // close the session
    session_write_close();
    header("location:/login_success.php");
    exit();
}
else
{
    echo "Wrong Username or Password";
}
?>
```

Of course, any page in a protected area is going to check that the requesting user has been authenticated, often using code similar to this.

```
<?php
// session_is_registered() is a function that was deprecated as
// of php 5.3.0 and should not be used; it's presence here is
// as an example of how [i]not[/i] to register session variables.

if (!session_is_registered(myusername))
{
    header("location:mainlogin.php");
}
?>
```

Effectively *myusername* can be empty and the auth script would see this as "logged in". The variable is registered but a better option would be to fill it with something concrete to check against. This is where our signature comes into play. By ensuring that user ID is present, *loggedIn* is true, and recalculating the signature and ensuring it matches what's stored in session data, we can be reasonably certain we're

dealing with a legitimate user.

```
<?php
if (!isset($_SESSION['user_id']) || !isset($_SESSION['signature']) || !
isset($_SESSION['loggedIn']) || $_SESSION['loggedIn'] != true ||
$_SESSION['signature'] != md5($_SESSION['user_id'] .
$_SERVER['HTTP_USER_AGENT'] . $salt))
{
    session_destroy();
    header("Location: mainlogin.php");
    exit();
}
?>
```

A note about database connections: We have included mysql_ examples in some of the above code snippets simply for the sake of familiarity. Since the release of PHP 5 way back in 2004, MySQLi has been the preferred method for working with MySQL databases and we strongly recommend moving away from the old mysql_ functions. An alternative to MySQLi, which affords the flexibility of using a DBMS other than MySQL without having to rewrite your code, is the Portable Database Object, PDO. In either of these cases, we feel it preferable to make use of prepared statements to protect against SQL injection.

Please note that the aim of this tutorial is simply to help users avoid some common pitfalls we've seen time and again. This is not the be all, end all of login tutorials, nor do we claim anything here is foolproof. We hope that this has been of use to those who don't know where to start when writing a login system. For those who are experienced in the matter please add any comments, critique or additional tips so the article can be as complete as possible.