Secure Website Login Programming with PHP & MySQL

If you are developing a web-based system whereby a user, or users, are logging in and staying logged in (sessions, cookies), the following ideas are written with you in mind.

Introduction
If you are developing a web-based system whereby a user, or users, are logging in and staying logged in (sessions, cookies), the following ideas are written with you in mind. Making sure your authentication and authorization schemes are secure is going to be part of your task. All of those things fall under the umbrella term: security. Any competent, security conscious person should already know that most intrusions/attacks are undertaken as follows:

1. Social Engineering (conning) - see the wikipedia definition
2. An inside job, by an employee or trusted person

What it all means is that nothing is stopping one of your users from choosing an easy password, sharing it with others, or leaving themselves logged in as they step away from the machine. Nor can you completely stop an employee from misusing your internal system. However, it behooves you to implement the most basic security measures in your programming, in this case, website programming. That is why I have written this article. There are many books and courses covering security, below is a list of further reading that I recommend:

- The Art of Deception: Controlling the Human Element of Security - ISBN 0471237124
- Computer Security for the Home and Small Office - ISBN 1590593162
- Building Secure Software - ISBN 020172152X
- Security Engineering - ISBN 0471389226

There are a few more books out there that are very helpful, though not listed here. I felt the ones above would give you the best head start.
Last, but not least, this article is a living, breathing document that most likely has errors. Do not hesitate to second guess anything below, and please email me with any changes, fixes or updates.

Some Basic Rules

Rule #1 -	**Nothing is totally secure. Break-ins and compromises are inevitable.**
Rule #2 -	**Segment your system/software in order to diminish the damage from said compromise.**
Rule #3 -	**Log as much as you can.**
Rule #4 -	**Never trust user input.**

My definition of security
Slowing down an attacker long enough to capture them, and/or fix the security holes, while at the same time safeguarding a system that is segmented in order to lessen the degree of damage during a successful attack. In other words, make a system that is designed for security, defense and facilitates recovery from attack. (Think like kevlar, not concrete: be flexilble, absorb attack, recover and respond.)

Basic Security Methods
The following should be in place in your system, as a minimum.

1. **Login names and passwords should be 6 characters long, or more**
2. **In the event of login failure, be very uncooperative**

Tell the user "Your login attempt was unsuccessful", not "Your password was missing the letter x" or "Your username is not in our system". Give very few leads as to why the login failed. They only serve to help intruders.

3. **Handle errors gracefully**
   Place the ampersat symbol (@) in front of many of your PHP function calls. If they fail, the ampersand will stop from from showing that failure in the browser window. This is very useful when making database calls but your database is down, or the SQL statement returns an error. Such messages would only give feedback to intruders, or look unprofessional to regular users.
4. **Passwords in the user account table of your database must be encrypted (SHA-1)**
   If someone were to somehow gain access to the database itself, and view all of the user accounts, they would be able to see logins, but not passwords. Unless they changed the password, which would alert the user once they realized they couldn't log in, or they tried to crack the encrypted password (possible, but hard) they would have no way of using their newly found information.
   To accomplish this, the "password" field in your SQL datbase should be 40 characters long, which will hold an SHA-1 encrypted string. Before you compare the user input password to the one stored in the database, use the PHP **sha1()** function to encrypt it.
   Example: **$encrypted = sha1($password);**
   Sample database data:
   Login name: bobsmith
   Password: d0be2dc421be4fcd0172e5afceea3970e2f3d940
5. **Never use "admin" or "root" as your adminstrator login name**
   Try to use something else, one that gives the same idea, but is more unique. Some examples are: superman, wonderwoman, allpower, etc...
6. **Log the total number of logins for each user, as well as the data/time of their last login**
   Logging the total is just a good indicator, and *may* be useful for security purposes depending on your system. Keeping track of their last login is very useful in the event that someone logged in using their account, without permission. You now know the time it happened, and if you log the date/time of any changes in your database and by whom, you can track what that intruder did while logged in.
   In order to accomplish the above, the user account table in your SQL database should have three extra fields:
   Logincount of type INTEGER
   Lastlogin of type TIMESTAMP (or datetime)
   Thislogin of type TIMESTAMP (or datetime)
   When the user logs in, in PHP, update that user's information in the database by incrementing their login count and by getting the timestamp using PHP's built in **date()** function. After successful login, first transfer the info stored in 'Thislogin' to the 'Lastlogin' field, and then insert the new date/time into 'Thislogin'.
7. **Strip backslashes, HTML, SQL and PHP tags from any form field data**
   If someone maliciously tries to send HTML, SQL or PHP code through a text field entry not meant to expect it, they can disrupt or break your code. Use the following PHP functions to strip out such text:
   strip_tags(), **str_replace()** and **stripslashes()**
   Example: **$login = @strip_tags($login);**
   Example: **$login = @stripslashes($login);**
8. **Add "LIMIT 1" to the end of your SQL statements**
   That will limit the number of results to just 1. If someone successfully hijacks your site, and is able to run a SQL statement that returns data, or deletes it, placing "LIMIT 1" at the end of any SQL string will help limit the amount of data they are able to see or damage.
   Example: **SELECT * FROM useraccount WHERE Login='$login' AND Password='$encrypted' LIMIT 1**
9. **Use the "maxlength" option in your HTML form elements**
   Limit the user to the allocated input size. If an login field in your SQL schema is of size 8 characters, limit the text field input to 8 using maxlength.
   Example: **<input type="text" name="login" size="8" maxlength="8">**
10. **Trim any and all form field data**
    Trim down the length of any form field data. If you expect a string of length 8, don't rely on the HTML maxlength (above), or the kindness of the user to pass you a string that long. Cut it down to size. Always.
    substr()
    Example: **$login = @substr($login, 0, 8);**

11. **Check the referrer**
    Make sure the login script checks the HTTP_REFERER to see where the request came from. It should come from your HTML form, on the same server. If not, reject the login attempt. Though, I must tell you the HTTP_REFERER is easy to "spoof", or fake, so this security measure is easy bypass. It will only stop simple spam bots, or the most amateur of attackers.
12. **Use $_POST not $_REQUEST**
    If your HTML form uses POST to send the data to the login script, then make sure your login script gets the input data using **$_POST**, and not **$_REQUEST**. The latter would allow someone to pass data via GET, on the end of the URL string.
13. **SSL Encryption (https)**
    To better ensure the privacy of the data being sent across the internet, purchase an SSL certificate to encrypt the login page, and any others.
14. **In general, limit user access according to their role**
    Design your system to give users specific layers, or subsets of access. Not everyone needs to be all powerful, nor all knowing. Using the unix group idea as your starting point. Classify users and give them features based on that. If you have a system with multiple users who have different roles, give them functionality based on those roles. Accountants, and only allow accountants can see financial data, not warehouse inventory or much else. The person at the cash register can enter in a sale, but not delete it. That is a managers job, and needs override permission. Etc....

<u>**More Extreme Methods**</u>
In page 1 of this article, I listed the most basic methods necessary for security. What will follow are some more extreme measures, which I refer to as "paranoid". I like that word, and in the security arena, it's a very good attitude to have. Don't mistake a paranoid security measure with a roadblock or hindrance. Though they can be one in the same, they don't have to be. What do I mean? For example, a paranoid company may tell you that the only time you can enter a server room is from 3:30 PM to 5:00 PM, one person at a time. Such a rule would no doubtedly hinder you're ability to do work. A counter example may be a paranoid company that logs *every* single failed login attempt into their system. It does not hinder your work, but sure does go above and beyond the most basic security.

<u>**Paranoid Methods**</u>

1. **Every login failure alerts an administrator**
   It doesn't have to be a siren going off, it could be a simple email detailing the date, time, IP address, and attempted login name.
2. **Store/log every user login**
   Instead of storing only the last login, and the current login of each user, create a table in your database soley for the storing of *all* logins. Give it fields to store the user name, password, date, time, and IP address.
3. **Third login failure disables the account, and/or disables by IP address**
   After three tries.....
   If someone fails to log in while using a valid login name, disable that account and alert an administrator.
   If someone tries to log in while using a login name not found in the system, log that IP address and block logins from that IP address.
   Note: For the first to be accomplished, your user account table needs to have a field called "disabled" of type TINYINT (to set it to 0 or 1) or ENUM (to store "Y" or "N").
4. **Use .htaccess and .htpasswd to double protect a site**
   In addition to a basic PHP login page that asks for authentication, put in place .htaccess and .htpasswd restrictions. It's pretty flimsy, but adds that little bit of extra security to make you feel safe at night.
5. **Authenticate by IP address, in addition to login and password**
   Not only should you authenticate a user by their login name and password, but you can also put in a third element: their IP address. If the user is always going to be logging in from the same computer (or subnet) you can also check their IP address to see if it matches one allowed by the system. This will protect you from someone trying to log in at an unauthorized location, such as from their home. Or, it will stop an outsider from using a login/password they got using devious means. However, as with *any* other security measure, this can be circumvented by spoofing an IP address. Don't let that stop you though. This paranoid method would

add a third wrench in the works of any intruder, making it just that much more difficult to break in.

6. **Create separate, role based MySQL accounts**
In page one of this article, I recommended you give users limited access depending on their role or job. The same should happen for the MySQL accounts your PHP code uses behind the scenes. For example, there should be a MySQL user account that is restricted to only SELECT access on the user account table. It's main purpose is to be used in the login authentication. Because the login page is the most visible to intruders, its parts are the most vulnerable. If intruders somehow find out the MySQL username/password in its PHP code, they may be able to use that to run their own SQL queries. For the other portions of your site, the same rules above apply. Create MySQL accounts with restricted access, depending on the code they are meant to be used in. Portions of your site that allow people to view data should internally use MySQL user accounts that only have SELECT access. Etc.... you get the idea.
7. **Use stored procedures (similar to user defined functions)**
In MySQL 4.0 (and below) there are functions that can created by using the CREATE FUNCTION statement. In MySQL 5.0 and above, there will be the ability to create stored procedures by using the CREATE PROCEDURE statement. If you create your own stored procedure to authenticate a login, you minimize the ability for someone to see the internal structure of your database. It also allows you to minimize the data returned, especially if someone is able to insert a malicious SQL statement into your code. However, using just stored procedures is not the end all solution. It should be use in conjunction with a limited-ability MySQL user account (see above).
8. **Gracefully handle CRITICAL failures**
This is more of an idea for exception handling than for security, but it can be for both. Often times a critical failure is the result of an intruder trying to do something they shouldn't be, which then "breaks" your site. Instead of dying (using the **die()** function) try something else.

   For example, many PHP programmers will do this:
   ```
   $result = @mysql_query($query, $db) or die("Could not get data");
   ```

   That's a good thing to do, it allows the code to die gracefully. However, the above will alert the intruder that the error was caught, giving them feedback and allowing them to try something different next time. Or, it will look *unprofessional* to a normal user when your site dies.
   Create your own function called "capturecritical()" and in addition to aborting all further processes, that function should log information and email it to an administrator. An improved example would then be:
   ```
   $result = @mysql_query($query, $db) or capturecritical("MySQL Query Error in XYZ.php, line 24",
   mysql_error(), $user, time());
   ```

   Your function would accept as varaibles a basic title, in this case the default "MySQL Query Error", then a more verbose description (in this case, the mysql_error() data), the user account that instigated it, and the date/time it happened. The function would then email an administrator with that data, or log it to a log file. It will also tell the user a critical error has occured.

Conclusion
You should now have a somewhat complete picture of what can be done to create a secure, login based site. Most of what I have discussed refers to programming, and your code. I have not discussed the finer points of security, which I briefly mentioned at the introduction and have to do with our most human failings. Outside of the scope of this article are additional security measures such as requiring your users to choose non-obvious passwords, forcing users to change passwords every 30 to 90 days, training them not to give out their password over the phone, and so on..... I will leave all of that to another article.

Always keep in mind, security is meant to slow down an attack enough for you to capture the intruder, or fend them off and then correct the security hole. If you think your site is 100% intruder proof, think again.
