Authentication Security

10 common mistakes you should look out for

1: not escaping request strings

This is one the most common errors, and unfortunately also one of the most dangerous.

Every string that goes into a SQL query (like the ones for handling user's data) must be escaped to avoid potential *SQL Injection attacks*.

Every PHP database extension has a way to escape strings: *mysqli* has *mysqli_escape_string()*, PDO does that in the *prepare() / execute()* methods and so on. Be sure to check the extension you are using.

Avoid relying on generic functions like *addslashes()* as they may not work correctly in all cases.

2: using plain text passwords

Passwords should be known only to their respective users. There is no need to store them anywhere. What you need is just a way to *check* if a password is valid.

You should definitely avoid storing plain text passwords. Instead, just store a secure hash of the password on the database, possibly using the PHP native functions *password_hash()* and *password_verify()* as explained in my tutorial.

This way, a database compromise will not allow the attacker to steal the account's passwords.

3: not using HTTPS

The web HTTP protocol itself is not secure because it sends the data unencrypted over the network. Everyone on the same network is therefore able to read your data and potentially execute “Main in the middle” attacks.

This is especially easy to do in private LANs.

By using HTTPS you will make all your data is unreadable to anyone except the destination server. HTTPS will also secure your cookies, one of the weak points of many authentication systems.

4: using weak encryption

Not every encryption mechanism has the same security. Too often, password hashes are created “manually” using some hashing algorithm like MD5 or SHA, and then the hashes are stored on the database as they are.

Some of these algorithms are inherently vulnerable (like MD5), but even those who are not are susceptible to dictionary attacks.

It’s much better to use more modern algorithms and use salted hashes, which dramatically lower dictionary attacks chances of success. Again, *password_hash()* and *password_verify()* take care of everything for you.

5: relying on “security through obscurity”

“Security through obscurity”, or *SOB*, is the idea that a system or a piece of code is secure as long as it remains secret.

For example, you could assume that some service cannot be exploited as long as nobody knows it exists or which parameters it needs to work.

Unfortunately, history has clearly proved this idea wrong and relying on SOB is a very bad idea.

Don’t assume your code is secure just because nobody knows how it works.

6: not securing Sessions

PHP Sessions are often used for authentication purposes.

That is ok, at least if you are not dealing with critical systems, but be aware that Sessions must be used with security in mind.

You need to avoid potential security problems like Sessions Hijacking.

I suggest you to read my Sessions Tutorial to understand how they work and how to properly secure them.

7: not enforcing password strength

The password itself is often a weak point.

If the password is easily guessable or can be found too quickly with brute force attacks, than any other security measure will be almost useless.

Be sure to enforce password strength: password should have a minimum length (at least 6 chars, but the more the better), letters with numbers and special symbols, and should be different from the username.

They should also be changed at regular intervals.

8: trusting request input data

Some web applications trust the request input data just because it's supposed to come from the website itself, like a login form or a AJAX connection.

This is a major error: request data can be easily forged, and client-side validation (like HTML or Javascript) can be easily circumvented as well.

Always check and validate all your input data as much as needed, and keep in mind that that data could come from an attacker trying to break your security.

9: not keeping user privileges separate

Your application should hide sensitive data from users unless they are authenticated, but that's just the first step.

More complex applications also need to verify *which user* is requesting the data, and check wether that specific user has the right privileges to do so.

This is called user privileges separation.

Failing to do so can let a user access data that should be accessible only to other users.

10: keeping login sessions open for too long

Usually, after a remote user is authenticated its “session” is left open for some time. This can be done with PHP Sessions or with custom login sessions mechanisms as well.

One common error is to keep those sessions open for too long, or worse indefinitely. If someone has stolen the user authentication cookie, then the account will be compromised until the current session is left open.

It’s a good idea to force the user to authenticate again after some time.