# Top PHP Secure Coding Practices for a Team

This whitepaper will discuss basic PHP secure coding practices to be followed by a team. PHP is a server-side scripting language that also has object oriented capabilities. It's one of the most common languages used for developing web applications. As a result, it's essential for a developer to understand how to write secure code in PHP. In this article, we will discuss how to write PHP code to protect from the most common category of attacks.

## Work together as a team

- Have a security master in the team to look at all security related changes. Accountability must be set at every level.
- Have a separate branch for all commits related to security and merge it into the master branch after a review by multiple team members. Also, keep a track of all the security related changes using a task tracker utility like Jira, Pivotal tracker etc.
- Have a document that documents the PHP secure coding standards. It is also important to make sure that you always stick to these standards. Spend time in updating those standards. These standards should also take into account the scope of the application, whether it's over Internet or intranet. Who are the users, is it internal users or external users (customers)? Who should have access to the application? What is the information it contains?
- Have a PHP security-testing checklist to validate that the security fix works. These tests should help eliminate the low hanging fruits and fix it during the development process.
- Have a list of certain libraries and common functions for fixing the most common type of vulnerabilities. Use the same functions for all the applications, so that if a change is needed, it is much easier to apply those changes across the entire code base. For example, to mitigate risks like XSS, CSRF etc., there should be common libraries and functions in place that should be used throughout the application.
- Make the new guy in the team go through the secure coding standards.
- Split the roles between multiple team members.
- Do security code reviews on a monthly basis.
- After the coding is complete and QA is done, go through another round of Vulnerability assessment on the application.

Secure coding as a team still requires adherence to good secure coding principles. Make sure to consider the following principles as you work together to ensure good coding practice.

### Protecting against file uploads

Unrestricted file uploads can pose a serious risk to a website can result in complete takeover of the web server. This is because an attacker can upload a malicious file on the web server if there are no checks on the server side. In addition, if it is possible to execute it by accessing the file path, it can allow him to execute his own payload and take control of the server. Hence, it is very important to keep a check on the files that are being uploaded. One of the common mistakes that developers do is check the file extension against a certain whitelisted extension of files. However, it is very trivial to bypass such kinds of restrictions. The following steps must be taken to ensure a secure file upload.

- Have a whitelist of allowed extensions and mime types of the file. While it doesn't eliminate the problem, but it does add an extra layer of security. Please note that it is very trivial to bypass this check because the web-server might use the first extension after the first dot (â€œ.â€) in the file name to identify the file extension. Therefore, it is possible to bypass this check by uploading a file with two extensions, for example, "shell.php.jpg”
- Have a limit on the file size of the uploaded file.
- Remove all the special characters in the filename before uploading it to the server.
- Change the name of the file before storing it on the server. Use some algorithm to determine a random name each time. This helps in the way that an attacker won't know the file path of the uploaded file if the file is renamed on the server, thereby not allowing him to access the file via the browser.
- Make sure that the files cannot be executed in the uploaded directory.
- Scan the file using a virus scanner after uploading it to the server. Malicious files should be removed.
- If possible, move the uploaded file in a directory outside of the web environment and use an internal script to download those files when requested. This makes sure that the attacker is not able to get control of the web application even if he is somehow able to execute the uploaded file.
- Make sure to apply all the above protections together for a better defense in depth approach.

## Protection against SQL injection

SQL injection occurs when user input forms part of the SQL query and could be manipulated to execute a different SQL query. For example, let's take the following SQL query that takes a user input (txtUserId) and puts it in the SQL query.

```
"SELECT * FROM Users WHERE UserId = " + txtUserId;
```

If a user is able to input the value of txtUserId as 11 or 1=1, we get the following query

```
"SELECT * FROM Users WHERE UserId = 11 or 1=1
```

This sql statement will effectively dump all the information from the users table

Using prepared statements is the way to go when protecting against SQL Injection attacks. The key feature of prepared statements is essentially in the way it treats the input parameters as different from the SQL query. The SQL statement is already created prior to receiving the input parameters and hence all the input is considered as only data and does not form a part of the SQL query. Please note that your database engine must support prepared statements in order for you to be able to use it. The following code snippet from PHP's official page is an example of how to use prepared statements.

```
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name,
:value)");
$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);

// insert one row
$name = 'one';
$value = 1;
$stmt->execute();

// insert another row with different values
$name = 'two';
```

```
$value = 2;
$stmt->execute();
```

## XSS protection

Cross Site Scripting (XSS) is one of the most common kinds of injection attacks found in web applications. It occurs when an attacker is able to execute Javascript code within the context of the victim's browser. For example in the url...

```
http://vulnerablesite.com?vuln_param=<script>alert(1)</script>
```

If the parameter vuln_param is output in the victim's code, it will essentially execute a script within the context of the victim's browser.

In order to prevent it, it is advisable to avoid getting inputs that can include HTML tags. PHP has a function *htmlspecialchars* that can be used for encoding the input. The way to do it is by using the following line of code.

*htmlspecialchars($string, ENT_QUOTES | ENT_HTML5, $charset);*

However it's recommended to use a Template engine throughout the whole application so you don't have to use this function all the time. These templating engines are one of the better ways of protecting against XSS attacks. A good one for php istwig.

## Other injection attacks

There are many other kinds of injection attacks that can happen by passing unsanitized user input other than XSS and SQL injection. The following in-built PHP functions can be used to perform proper input sanitization.

- htmlspecialchars - Convert special characters to HTML entities

  ```
  $new = htmlspecialchars("Test", ENT_QUOTES);
  echo $new; // <a href='test'>Test</a>
  ```

- htmlentities - Convert all applicable characters to HTML entities
- strip_tags - Strip HTML and PHP tags from a string

## CSRF protection

It is important to protect against every request that does something noteworthy (change email, change password, update user information) from CSRF attacks. It is important to note that if the developer is relying on some information which is being sent HTTP headers, cookies etc. for authentication, in that case CSRF protection also needs to be applied. This can be done using a unique CSRF token for each request being made and then validating that unique token on the server side when the request is being sent. Apart from this, every sensitive request should have some kind of re-authentication in place. Apart from this, adding Captcha is another way of protection against CSRF attacks. To get an idea on how to protect against such CSRF attacks in PHP, it is highly recommended to check out this code snippet from OWASP. CSRFMagic is a library that can be used to protect against these attacks.

## Session Management

Proper session management is one of the key things to have in a secure web application.

1. Make sure that the generated Session ID has significant entropy to prevent it from being brute forced. The default method in PHP for generating session id is dependent on the IP address, current time, a pseudo random number generator (PRNG) and an OS-specific random source. While it guarantees a lot of randomness, it is not completely secure. It is also recommended to use your own algorithms to generate Session IDs, as long as they provide enough randomness. The following PHP code snippet will generate a random session id.

```
session_start();

$old_sessionid = session_id();

session_regenerate_id();

$new_sessionid = session_id();

echo "Old Session: $old_sessionid";

echo "New Session: $new_sessionid";

print_r($_SESSION);
```

2. Make sure there is a valid session timeout.
3. If you are using your own function to generate Session ID, make sure it is mapped to an IP address, device ID etc to prevent against Session fixation attacks. The IP of the user can be fetched by the following command *getenv ( "REMOTE_ADDR" );*
4. Session IDs passed in headers should be passed over a secure TLS connection and should have the *HTTPOnly* and *Secure* flag set.

```
      void session_set_cookie_params ( int $lifetime [, string $path [,
string $domain [, bool $secure = true [, bool $httponly = true ]]]] )
```

5. Session IDs should change with changes in privilege.

## Proper password protection

Passwords should not be stored in plain text on the database. Use proper hashing on the passwords before storing them in the database. Encryption is not recommended while storing passwords on the database because it can be decrypted back to its original value if the attacker knows the algorithm and key, whereas hashing is much difficult to break because the only way to break a hash is using brute force techniques. PHP supports many hashing algorithms and they should be used with a proper Salt to introduce even more randomness. The below code snippet gives an example on hashing passwords. If possible, use multiple hashing algorithms to hash the password multiple times before storing in the database. The following code snippet shows how to hash a password with a randomly generated salt.

```
/**
 * Note that the salt here is randomly generated.
 * Never use a static salt or one that is not randomly generated.
 *
```

```
 * For the VAST majority of use-cases, let password_hash generate the
salt randomly for you
 */
$options = [
    'cost' => 11,
    'salt' => mcrypt_create_iv(22, MCRYPT_DEV_URANDOM),
];
echo password_hash("rasmuslerdorf", PASSWORD_BCRYPT, $options)."\n";
```

## Error Handling

Error message can disclose sensitive information about the application. For example, let's look at the following error.

```
Warning: opendir(Array): failed to open dir: No such file or directory
in C:\Users\bob\public_html\includes\functions.php on on line 84
```

The following error discloses the file path of the application and also informs of a user named Bob on the system, which can later be used for brute force attacks. It is important to make sure that your application doesn't give much information away in case of an error. Hence, it is recommended to turn off all error reporting in a live application. This can be done by editing the .htaccess or php.ini, by setting "error_reporting" to "0". It is also recommended to have a default error page for all kinds of errors. For example, the following error page shown by Facebook is an example of a generic error page that doesn't disclose any sensitive information.


## Protection against remote code execution

There are certain functions in PHP (shell_exec, exec, passthru, system) that can execute commands on the server. If an attacker can get user input to go as input to these functions, it can lead to remote code execution attacks. Let's look at the following piece of code that takes an input with the parameter name and outputs it back to the user.

```
<?php
$name = $_GET['name']
system("echo $name");
?>
```

An input like *Hacker && dir* will essentially execute the directory command on the server and output the result back to the user. Hence, it is very important to perform proper input sanitization on these requests and make sure that you never pass manipulated user input to these functions.

### Other tips and tricks

1. Use of $_REQUEST is not recommended at all, as it allows the use of methods like COPY, DELETE etc. It is always recommended to specify the exact method to be used.
2. Do not trust third party libraries. Perform a thorough source code review on them before using in your application.
3. Edit the *.htaccess* file to limit access to directories that hold your configuration files.
4. Do not store backup files with the extension .bak as they can be used to download these files by directly accessing their path.
5. The default security features in PHP such as *addslashes, mysql_escape_string,mysql_real_escape_string* should be enabled in the php.ini file.
6. *Register_Globals* variable in php.ini should be set to OFF to protect against file inclusion and variable poisoning attacks.
7. All default passwords in Mysql, third party libraries etc must be changed.

## References


1. PHP official page - http://php.net
2. OWASP - https://owasp.org