Best Practices for Writing Secure PHP Code

Vandit H. Shah, Neel J. Suthar, Aman K. Sharma, Manasa Sarjana, and Sagarika Veranki

School of Continuing Studies, York University

CSCO1070 – Cybersecurity Operations Capstone

April 08, 2024

Abstract

In this study, we looked at secure coding techniques for PHP-based web applications to help prevent cyber threats and protect sensitive data. Our study examined a variety of strategies, including user input sanitization, input validation techniques, robust authentication systems, and safe configuration procedures. We also looked into encrypted communication, secure testing methodologies, error handling, and file processing. Our findings underscore the need of following these measures to improve PHP application security by assuring data confidentiality, integrity, and availability. It is critical that enterprises follow these standards in order to retain user trust and confidence in the digital ecosystem.

Keywords: PHP security, web application security, secure practices, securing PHP

Contents

List of Figures

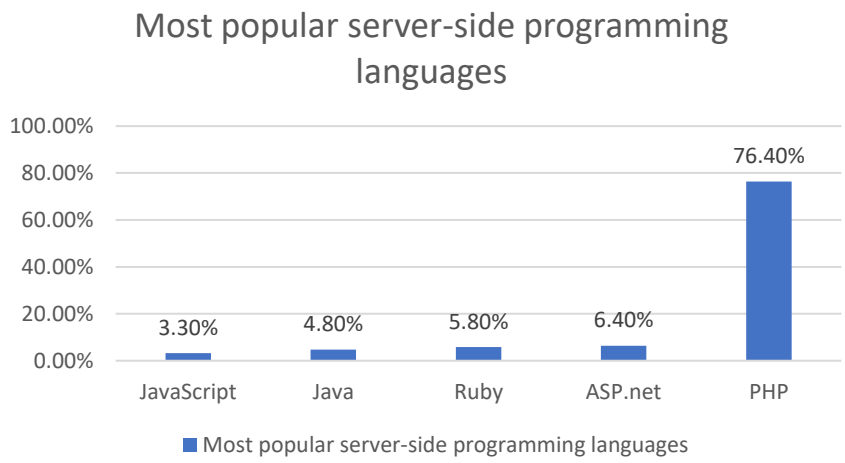
Introduction

In today's digital age, PHP-based web apps have become increasingly popular, acting as critical platforms for delivering services and communicating with clients (Shiflett, 2005). However, rising usage raises the potential of cyber assaults targeting vulnerabilities in these programs. Recognizing the critical relevance of protecting PHP applications, this article investigates the use of secure coding approaches to reduce these risks and protect sensitive data.

The inspiration for this study stems from the crucial necessity to address the rising cyber risks to PHP applications. We hope to improve the security posture of these applications by using secure coding methods that ensure data confidentiality, integrity, and availability. We aim to give actionable insights for stakeholders and developers by conducting a complete examination of numerous methodology, such as user input sanitization, input validation techniques, robust authentication systems, and safe configuration procedures.

Figure 1

Data of most popular server-side programming languages



Note. Data from “Usage statistics of server-side programming languages for websites” by W3Techs as of March 31, 2024

Our approach entails investigating real-world circumstances and best practices, depending on credible sources such as OWASP, the PHP Manual, and the SANS Institute. By emphasizing the importance of secure coding techniques, we hope to provide organizations with the information and tools they need to navigate the expanding threat landscape while maintaining user trust and confidence in the digital ecosystem.

Problem Statement

The goal of this report is to provide complete guidelines for implementing secure coding standards in PHP. By implementing these practices, new developers, and businesses with no dedicated security team will get the knowledge and awareness required to protect PHP applications against common cyber risks. A totally secure system is a virtual impossible, hence a common approach in the security profession is one of balancing risk and usability (*PHP: General Considerations - Manual*, 2024). The major weakness in many PHP systems is not inherent in the language itself, but rather a lack of security-conscious coding (*PHP: User Submitted Data - Manual*, 2024). The major goal is to improve the robustness of PHP applications, reduce vulnerabilities, and protect sensitive data, encouraging trust and confidence among users and business.

PHP applications are vulnerable to a various types of cyber-attacks, including SQL injection, Cross-Site Scripting (XSS), and remote code execution. CyberNews analysts discovered more than 80,000 servers globally are still running on older versions of PHP, which are exposed to hundreds of known vulnerabilities, making them easy prey for threat actors (*Unpatched and Unprotected*, 2021). These vulnerabilities can result in unauthorized access, data breaches, and other security incidents that harm an organization's brand.

This report aims to develop a free educational report to enhance PHP web developers' knowledge of secure coding techniques. PHP being the most widely used backend

programming language, this tool has the potential to reach a vast audience. By fostering a greater emphasis on security among developers, it can ultimately strengthen website security. This whole report will elaborate on the report's purpose, requirements, and objectives, delve into secure coding guidelines and tools, and propose the methods and techniques to be followed to supplement existing security resources for PHP developers.

Secure Coding Practices

Secure coding involves developing code that follows best practices for code security. This practice helps protect against known, unknown, and unexpected vulnerabilities, such as security exploits, data leaks, and identity theft. By implementing secure coding, we can minimize attack vectors and enhance the security of our applications. Here are the seven secure coding practices which a developer and business can benefit from.

Sanitize User Input

Currently to secure any systems, Zero Trust (ZT) has become critical asset for any organization. Zero Trust is basically based on not trusting any request or query for accessing critical resources. Because of widely usage of cloud, Systems may now be accessed from a wider range of devices and places. At the same time, attackers have become more frequent (Fernandez & Brazhuk, 2024). In brief, ZT means “Trust No One”. The NIST standard framework defines Zero Trust as a comprehensive approach to data security, covering identity, credentials, access management, operations, endpoints and hosting environments (Rose et al., 2020).

By implementing zero trust policy in sense of sanitizing user input, the developer mitigate the risk of various security threats such as SQL injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF). Adversaries often exploit vulnerabilities in web applications by injecting harmful code or tricking servers into executing unintended

actions. Here zero trust policy works as a proactive defence mechanism against such attacks by ensuring that all user inputs are sanitized before processing the request or the data.

htmlspecialchars()

This function converts special characters to HTML entities. It prevents XSS attacks by encoding characters like <, >, ", ', and &.

Figure 2

Example of Implementing htmlspecialchars()

```
<?php

$input = "<script>alert('XSS attack by Vandit!');</script>";

$secure_input = htmlspecialchars($input);

echo $secure_input;

//Output &lt;script&gt;alert(&#039;XSS attack by

Vandit!&#039;);&lt;/script&gt;

?>
```

htmlentities()

Similar to htmlspecialchars(), this function converts all applicable characters to HTML entities. It's useful for ensuring all characters are properly encoded, especially in scenarios where htmlspecialchars() might not cover all cases.

Figure 3

Example of Implementing htmlentities()

```
<?php

$$input = "<script>alert('XSS attack by Vandit!');</script>";
```

```
$secure_input = htmlentities($input);

echo $secure_input;

//Output &lt;script&gt;alert(&#039;XSS attack by

Vandit!&#039;);&lt;/script&gt;

?>
```

strips_tags()

This function removes all HTML and PHP tags from a given string. It's often used to prevent XSS attacks by stripping potentially malicious code from user input.

Figure 4

Example of Implementing strips_tag()

```
<?php

$input = "<p>Test paragraph.</p><!-- Test Comment --> <a

href='#YorkUniversity'>Other text</a>";

$secure_input = strip_tags($input);

echo $secure_input;

// Output: Test paragraph. Other text;

?>
```

real_escape_string()

Specifically designed for database operations, this function escapes special characters in a string for use in an SQL statement, thus preventing SQL injection attacks.

Figure 5

Example of Implementing real_escape_string()

```php
<?php

// Assuming $mysqli is your database connection

$input = "Neel's SQL injection attempt";

$secure_input = $mysqli->real_escape_string($input);

$query = "SELECT * FROM users WHERE name='$secure_input'";

?>
```

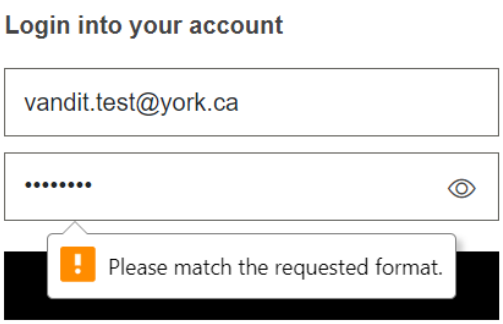
Validation of User Input:

PHP is a user friendly server site scripting language. PHP's relaxed attitude toward variables (allowing them to be used without having been declared, and converting types automatically) is ironically an open door to possible trouble (Powers, 2009). Never use the user input as it is, always validate the user input before implementing it.

The PHP Filter extension provides a set of functions for filtering and validating user input. It allows you to sanitize user input to remove any harmful characters or code and validate user input to ensure that it meets specific criteria. By using the PHP Filter extension to sanitize and validate user input, you can ensure that your PHP code is secure and user inputs are validated properly (*PHP: User Submitted Data - Manual*, 2024).

Figure 6

Basic example of Password field Input Validation in login form



Validation using regex

Regular expressions provide a powerful way to validate user input. For example if we take email as an user input and it can be checked using function called preg_match(). This function checks if email follows specific pattern or not.

Figure 7

Example of Implementing preg_match()

```
<?php

// User input (email address)

$email = "yorkstudent@yorku.ca";

// Regex pattern for validating email addresses

$email_pattern = '/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/';

// Validate email using preg_match()

if (preg_match($email_pattern, $email)) {

    echo "Valid email address: $email";

} else {

    echo "Invalid email address: $email";

}

// This can be implemented whenever we are execpting input from user

// Always validate the input whether it is email, password or name.



?>
```

Validation using PHP filters

Built-in filter functions allows you to validate different types of user input. If we take user email as user input and can validate it using filter_var() with FILTER.VALIDATE.EMAIL filter.

Figure 8

Example of Implementing filter_var() with FILTER.VALIDATE.EMAIL filter

```
<?php

// User input (email address)

$user_email = "yorkstudent @yorku.ca";

// Validate email using filter_var() with FILTER_VALIDATE_EMAIL filter

if (filter_var($user_email, FILTER_VALIDATE_EMAIL)) {

    echo "Valid email address: $user_email";

} else {

    echo "Invalid email address: $user_email";

}

// This can be implemented whenever we are execpting input from user

?>
```

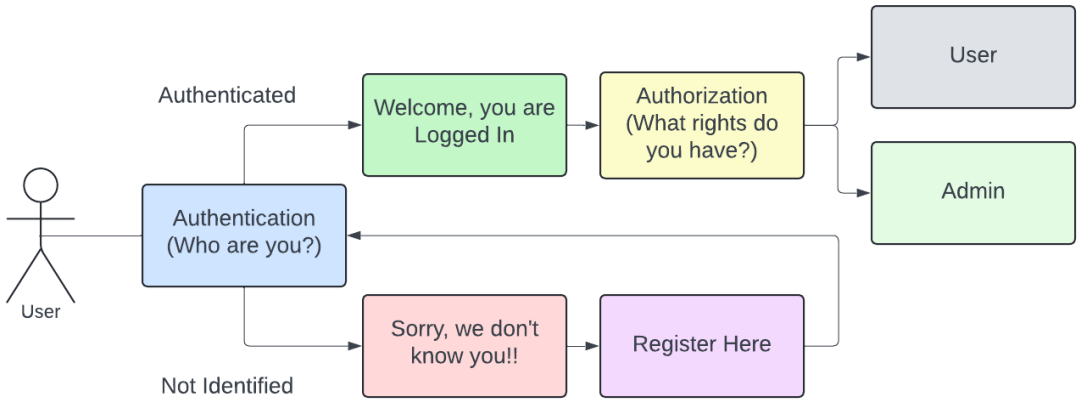
Authentication and Authorization

Authentication is generally used to prove user's identity. The basic example would be username and password check. While Authorization is generalized as an access control, to determine how a user is authorized to use or access the particular resources. Access controls

needs to be integrated in web applications to prevent attacks. It can be implemented through Access control, and it also need a reliable identification mechanism (Shiflett, 2005).

Figure 9

Concept of Authentication and Authorization



In PHP, basic authentication contains simple access controls such as username and password as a plain text, for that HTTP 1.1 contains more secure method known as “Digest Authentication” which uses a MD5 hashing algorithm to encrypt these details (Welling & Thomson, 2009). By simply implementing any of access control mechanism, we can avoid common concerns such as brute-force attacks, password sniffing and replay attacks for web applications (Gilmore & Treat, 2006).

Password Hashing and Salting

Passwords should never be stored in plaintext to prevent exposure in the event of a data breach. Instead, they should be securely hashed and salted. PHP's password_hash() function is used to hash passwords using a strong cryptographic algorithm, while password_verify() is employed to verify hashed passwords during authentication. It is not recommended to use salt() to explicitly provide salt. By default random salt will be generated

by password_hash() for each password hashed. The salt option has been deprecated. It is now preferred to utilize the salt generated by default. As of PHP 8.0.0, an explicitly specified salt is disregarded (*PHP: General Considerations - Manual*, 2024).

Figure 10

Concept of Password Hashing and salting

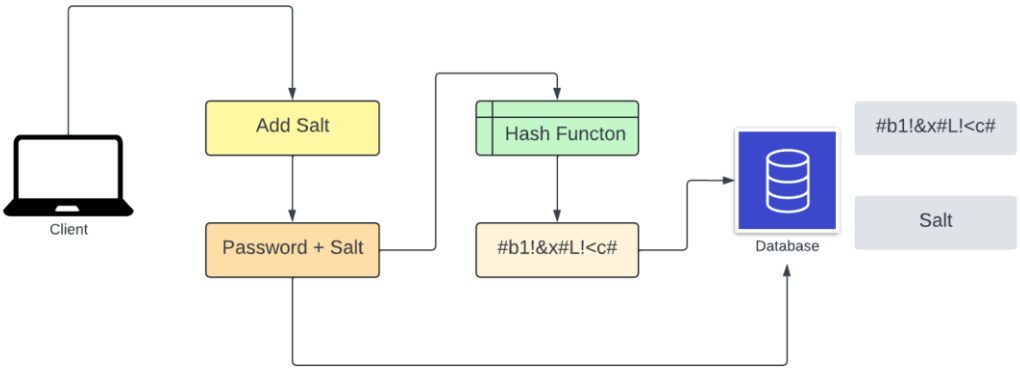


Figure 11

Example of Implementing password_hash() and password_verify()

```
<?php

// User's password

$user_password = "vandit_password_123";

// Hash and salt the password

$hashed_password = password_hash($user_password, PASSWORD_DEFAULT);

// Store $hashed_password in the database

// Later during authentication, verify the password
```

```
$user_input_password = "vandit_password_123"; // Password entered by the
user during login

if (password_verify($user_input_password, $hashed_password)) {

    // Password is correct

    echo "Password is correct";

} else {

    // Password is incorrect

    echo "Password is incorrect";

}

?>
```

Role Based Access:

Users should only be granted access to resources and functionalities that are necessary for their roles or tasks. We should define the roles of users in schema of database and when the user is stored it should be assigned a role. When user log into the system before fetching content as whole from backend we should first verify the role of the user. Based on the role the user should be permitted to the content he is authorized to.

Figure 12

Example of Implementing Role Based Access

```
<?php

// Check user's role or permissions before granting access

$user_role = "admin"; // Assume the user's role is "admin"

// Example of enforcing authorization based on user role
```

```
if ($user_role === "admin") {

    // Grant access to admin-specific functionalities

    echo "Welcome, Admin!";

} else {

    // Deny access

    echo "Unauthorized access";

}

?>
```

Secure Configuration

Secure configuration in PHP makes sure that flow of information is secure at each point. As a network security perspective, security in depth is a key. So never rely on single protection method for your PHP code. A multilayer approach which involves server, PHP code, files , databases etc. is what actually need to be implemented (Ballad & Ballad, 2009).

One of the capabilities of PHP is its native session management which give some tools to developers to create a secure PHP environment. In brief, secure the network, secure the Server and secure the application. It is essential to secure PHP configuration to counter software vulnerabilities and poorly protected data to safeguard against attackers. One of the most common ways to ensure secure configuration is to keep updated your PHP version. Moreover, PHP offers a number of configuration parameters such as SAFE_MODE in shared server environment (Gilmore & Bryla, 2007).

Disabling register_globals

The register_globals directive, when enabled, allows incoming parameters to

automatically create global variables, posing a significant security risk. It should be disabled to prevent potential injection attacks.

Figure 13

Disabling register_globals

```
// In php.ini configuration file

// Find where register_globals variable is and disable it

register_globals = Off
```

Disabling allow_url_include

The allow_url_include directive allows including files from remote URLs, which can be exploited by attackers to execute arbitrary code on the server. It should be disabled to mitigate remote code execution vulnerabilities.

Figure 14

Disabling allow_url_include

```
// In php.ini configuration file

// Find where allow_url_include variable is and disable it

allow_url_include = Off
```

Keeping PHP Updated

Regularly updating PHP to the latest version is crucial for maintaining security. This ensures that known vulnerabilities are patched and security features are up-to-date.

Figure 15

Updating PHP versions in Linux Server with Command Line

```
# Command-line update using package manager (e.g., apt-get for Debian-
based systems)

$ sudo apt-get update

$ sudo apt-get upgrade php
```

Secure Communication

Most of the attacks are happened during transmission phase so it is required to make sure that PHP applications are safeguard against such violating entity. Manipulation of data passed via Web forms, URL parameters, cookies, and other readily accessible routes enables attackers to strike the very crucial part of your PHP code (Welling & Thomson, 2009). Using HTTPS and OpenSSL functions, we can simply avoid Man-In-the-Middle and eavesdropping attack.

Enabling HTTPS

HTTPS encrypts data exchanged between the client (e.g., web browser) and the server, ensuring secure communication over the internet. To enable HTTPS, an SSL/TLS certificate must be installed on the server. Earlier HTTP was used but it is very unsecure. While providing URL to action attribute in form tag, make sure HTTPS protocol is defined.

Figure 16

Example of Using HTTPS protocol for data transfer

```
<!-- HTML code for a form submission -->

<form action="https://group-a.com/process.php" method="post">

    <!-- Form fields -->

</form>
```

Encrypting Data

PHP's openssl_encrypt() function can be used to encrypt sensitive data before transmitting it over the network. It supports various encryption algorithms and modes.

Figure 17

Example of Implementing OpenSSL Encryption

```
<?php

// Data to be encrypted

$data = "YorkU Sensitive Student Information";

// Encryption key and initialization vector (IV)

$key = openssl_random_pseudo_bytes(32); // 256-bit key

$iv = openssl_random_pseudo_bytes(16); // 128-bit IV

// Encrypt the data using AES encryption algorithm (256-bit key) and CBC
mode

$encrypted_data = openssl_encrypt($data, 'aes-256-cbc', $key,
OPENSSL_RAW_DATA, $iv);

// Base64 encode the encrypted data for safe transmission

$encrypted_data_base64 = base64_encode($encrypted_data);

// Transmit $encrypted_data_base64 over HTTPS

echo "Encrypted Data: $encrypted_data_base64";



?>
```

Decrypting Data

PHP's openssl_decrypt() function can be used to decrypt encrypted data received from the client. It requires the same encryption key and initialization vector (IV) used for encryption.

Figure 18

Example of Implementing OpenSSL Decryption

```
<?php

// Encrypted data received over HTTPS - Taking data from previous example

$encrypted_data_base64 = "encrypted_data_here";

// Decode the base64-encoded encrypted data

$encrypted_data = base64_decode($encrypted_data_base64);

// Decrypt the data using the same encryption key and IV

$decrypted_data = openssl_decrypt($encrypted_data, 'aes-256-cbc', $key,
OPENSSL_RAW_DATA, $iv);

echo "Decrypted Data: $decrypted_data";

//Output= Decrypted Data: YorkU Sensitive Student Information

?>
```

Error Handling and Logging

By default all the errors are thrown to user and these errors can reveal information that could be useful to malicious users who are trying to get into your application (Cosentino, 2003). To prevent the disclosure of sensitive information it is crucial to implement effective

error handling mechanisms. Securely log errors to a location inaccessible to attackers for analysis and troubleshooting purposes.

Error Handling

Set the appropriate error reporting level in PHP configuration (php.ini) or PHP code using error_reporting () function. The error_reporting() function specifies which errors are reported. PHP has many levels of errors, and using this function sets that level for the current script. We can use a customer error handler using set_error_handler() to catch PHP errors and exceptions and handle them gracefully.

Figure 19

Example of Different Error Level Reporting

```
<?php

// Turn off all error reporting

error_reporting(0);

// Report simple running errors

error_reporting(E_ERROR | E_WARNING | E_PARSE);

// Reporting E_NOTICE can be good too (to report uninitialized

// variables or catch variable name misspellings ...)

error_reporting(E_ERROR | E_WARNING | E_PARSE | E_NOTICE);

// Report all errors except E_NOTICE

error_reporting(E_ALL & ~E_NOTICE);

// Report all PHP errors

error_reporting(E_ALL);
```

```
// Report all PHP errors

error_reporting(-1);

// Same as error_reporting(E_ALL);

ini_set('error_reporting', E_ALL);

?>
```

PHP allows developers to define custom error handlers using the set_error_handler() function. Custom error handlers can catch PHP errors, warnings, notices, and exceptions, allowing developers to handle them gracefully

Figure 20

Example of Implementing set_error_handler()

```
<?php

// Custom error handler function

function customErrorHandler($errno, $errstr, $errfile, $errline) {

    echo "<b>Error:</b> [$errno] $errstr<br>";

    echo "Error on line $errline in $errfile<br>";

}

// Set custom error handler

set_error_handler("customErrorHandler");

// Trigger a PHP notice

echo $undefined_variable;

?>
```

Error Logging

Error logging in PHP is crucial for monitoring application health, diagnosing issues, and debugging errors. PHP provides the error_log() function, which allows developers to log errors to a file or system logger. Error logging provides valuable insights into the runtime behaviour of PHP applications. By logging errors, warnings, notices, and exceptions, developers can track down issues, identify trends, and troubleshoot problems efficiently. This proactive approach to error management improves application reliability and helps maintain a high level of user satisfaction (Cosentino, 2003; *PHP: General Considerations - Manual*, 2024).

Figure 21

Example of Implementation of error_log()

```
<?php

// Log a message to a file

$error_message = "An error occurred: Unable to connect to the database";

$error_type = E_USER_ERROR; // Error type (user-generated error)

$error_log_file = "error.log"; // Destination file path

// Log the error to the file

error_log($error_message, $error_type, $error_log_file);

?>
```

Secure File Handling

Securing file handling in PHP is important to prevent unauthorized access to sensitive files as well as to restrict file permissions by implementing access controls. PHP contains

inbuilt functions to create, read, write and manipulate files and directory. The common approach is to implement file access control for particular user using chmod(), chown() and chgrp() methods.

Setting File Permissions

The chmod() function in PHP is used to set the file permissions (mode) of a file or directory. Developers can specify the desired permissions using numeric or symbolic notation.

Figure 22

Example of Setting File Permissions

```
<?php

// Read and write for owner, nothing for everybody else

chmod("/york/data/file.txt", 0600);

// Read and write for owner, read for everybody else

chmod("/york/data/file.txt", 0644);

// Everything for owner, read and execute for others

chmod("/york/data/file.txt", 0755);

// Everything for owner, read and execute for owner's group

chmod("/york/data/file.txt", 0750);

// We use this file permission according to the use case



?>
```

Changing File Ownership and Group

The chown() function is used to change the owner of a file, while the chgrp() function is used to change the group ownership. Developers can use these functions to assign specific users or groups ownership of files or directories.

Figure 23

Example of Changing File Ownership and Group

```
<?php

// Change file ownership

$file_path = "/york/data/file.txt";

$user = "username"; // New owner username

// Change the owner of the file

chown($file_path, $user);

// Change file group ownership

$group = "groupname"; // New group name

// Change the group ownership of the file

chgrp($file_path, $group);

?>
```

Conclusion

To summarize, this white paper has offered a thorough assessment of the critical requirement for secure coding strategies in PHP applications to prevent cyber threats and protect sensitive data. Stakeholders and developers are provided with critical tools to improve security by investigating various methodologies such as user input sanitization, validation techniques, robust authentication mechanisms, secure configuration practices, encryption for secure communication, and error handling. The main message from this study is the crucial relevance of following these recommendations to assure data security, integrity, and availability, creating trust and confidence among users and stakeholders in the field of technology. Organizations can effectively reduce common cyber threats and vulnerabilities by establishing proactive defense mechanisms and following the best practices suggested in this document, thereby preserving their digital assets and reputation in today's interconnected world.

Resources

In this section, you can find materials related to secure coding practices for PHP applications. For access to the codes discussed in this white paper and further resources, please visit our GitHub repository: GitHub Repository Link. This repository contains sample code snippets, implementation guidelines, and the full text of this white paper for easy reference and accessibility.

References

Ballad, T., & Ballad, W. (2009). *Securing PHP web applications: Easy, powerful code security techniques for every PHP developer*. Addison-Wesley.

Cosentino, C. (2003). *Advanced PHP for Web professionals*. Prentice Hall PTR.

Fernandez, E. B., & Brazhuk, A. (2024). A critical analysis of Zero Trust Architecture (ZTA). *Computer Standards & Interfaces*, *89*, 103832. https://doi.org/10.1016/j.csi.2024.103832

Gilmore, W. J., & Bryla, B. (2007). *Beginning PHP and Oracle: From novice to professional*. Apress.

Gilmore, W. J., & Treat, R. H. (2006). *Beginning PHP and PostgreSQL 8: From novice to professional ; [learn how to build dynamic, database-driven web sites with two of the world's most popular open source technologies]*. Apress.

PHP: General considerations—Manual. (2024, April 1). https://www.php.net/manual/en/security.general.php

PHP: User Submitted Data—Manual. (2024, April 1). https://www.php.net/manual/en/security.variables.php

Powers, D. (2009). *The essential guide to Dreamweaver CS4 With CSS, Ajax, and PHP*. Friends of Ed Distributed by Springer-Verlag New York.

Rose, S., Borchert, O., Mitchell, S., & Connelly, S. (2020). *Zero Trust Architecture*. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-207

Shiflett, C. (2005). *Essential PHP Security: A Guide to Building Secure Web Applications*. O'Reilly Media, Inc.

Unpatched and unprotected: More than 80,000 PHP servers are vulnerable to cyberattacks. (2021, May 5). Cybernews. https://cybernews.com/security/unpatched-and-unprotected-more-than-80000-php-servers-are-vulnerable-to-cyberattacks/

Welling, L., & Thomson, L. (2009). *PHP and MySQL Web development* (4th ed). Addison-Wesley.