**Vulnerability Mitigations in Web Applications with PHP**

Alexandre Paquette

Department of Computing – South East Technological University

ZDEVC4203 – Secure App Development

Richard Butler

Date Submitted: April 12th, 2024

# Executive Summary

This report delves into security threads present in web applications using PHP and outlines mitigations for each identified issue. A simple web app with pre-existing vulnerabilities was used as an example for this report. The primary focus is on common security threats such as Cross-Site Scripting (XSS), Session management flaws, SQL Injection (SQLi), Secure Password Storage, Cross-Site Request Forgery (CSRF), Directory Traversal, and Brute-Force Attacks. The report provides insights into the risks and necessary countermeasures.

Cross-Site Scripting (XSS) vulnerabilities pose significant risks to web applications. They are categorized as reflective, persistent, and DOM-based. This report highlights an identified reflective XSS vulnerability and provided a sanitization method to mitigate it.

SQL Injection (SQLi) risks are mitigated by reworking query processing methods to ensure parameterization, preventing unauthorized access and manipulation of database contents.

Secure Password Storage involves hashing and slating techniques. They are required at registration, login, and password change to protect user credentials.

Cross-Site Request Forgery (CSRF) vulnerabilities in password reset functionality is mitigated by implementing a CSRF token and managing session lifecycles effectively.

Directory Traversal vulnerabilities allow unauthorized access to server files. They are mitigated by implementing stringent input sanitization checks and restricting file access to designated directories.

Lastly, Brute-Force Attack vulnerabilities in registration processes are addressed by implementing IP address lockout mechanism to prevent repeated failed registration attempts. The same mechanism is already in place for multiple login attempts.

# Cross-Site Scripting

Cross-Site Scripting is a security vulnerability that allows attackers to inject malicious code into web pages. The code can be used to execute malicious actions on behalf of the attacker. Cross-Site Scripting vulnerabilities fall into three categories: reflective, persistent, and DOM-based.

Reflective XSS involves injecting the malicious script into the URL, and its contents are reflected by the web application. This type of attack relies on social engineering to trick users into clicking malicious links.

Persistent XSS involves storing the input into a website’s database for use in a later request. When other users visit the affected web page, the server retrieves the stored script and delivers it to the browser, executing the malicious code within the context of the legitimate site.

DOM-based XSS involves injecting a payload that is manipulated by JavaScript inserted into the DOM. Unlike reflective or persistent XSS, DOM-based XSS is executed on the client-side and is not detectible by the server.

## Vulnerability Identified: Reflective XSS

In the web application, a Reflective XSS vulnerability was identified. At the index page, the username is being reflected in the browser on a failed login. A sanitization method was implemented and is used whenever user input is received.

**Location**: index.php

**HTTP Request**: POST

**Parameter**: Username

**Payload**: <script>alert(1)</script>

### Code Fix



# Session Management

Session Management involves securely handling user sessions within a web application. During the session, the user should only be allowed to access certain pages while authenticated. If the user becomes un-authenticated, they should no longer have access to authenticated pages.

## Vulnerability Identified: Improper Session Termination

When the user attempts to log out, their session remains active, keeping them logged in. This means they maintain access to the entire application even when attempting to log out. A session termination mitigation was implemented into the logout.inc.php page to properly terminate the session.

**Location**: header.php

**HTTP Request**: POST

**Behaviour**: User clicks the logout button

### Code Fix



*Located in the logout.inc.php file*

## Vulnerability Identified: Invalid Admin Access

Non-administrator users can access the admin.php page. Even if the option is not visible in the navigation menu, a user can navigate directly to admin.php via the url. A mitigation was implemented in the admin.php page to check if the user accessing the page, is an administrator.

**Location**: admin.php

**HTTP Request**: POST

**Behaviour**: User manually navigates to admin.php via url

### Code Fix

****

*Located in admin.php*

## Vulnerability Identified: Improper Session Management

After the session is terminated, the user is still able to view secure pages by hitting the back button. A session validation method was created to check if a user is authenticated when trying to access the pages. A unique token is also generated when the user logs in and is stored to validate that they are still authenticated.

**Location**: admin.php, auth1.php, auth2.php

**HTTP Request**: POST

**Behaviour**: User hit back button in browser to navigate to previous pages

### Code Fix



*Located in LoginManager.php, in ProcessLogin method*



*Gets invoked in auth1.php, auth2.php, and admin.php*

# SQL Injection

SQL injection involves manipulating SQL queries to execute unintended commands. An attacker can interfere with SQL queries sent to the database to gain unauthorized access, manipulate the database contents, or execute commands on the server.

## Vulnerability Identified: SQLi

At the login page, an SQL command can be injected when providing login credentials. To fix this vulnerability, I’ve reworked how queries are processed with the database to ensure all queries are parameterized. The new method has replaced all previous queries executed on the database.

**Location**: login.inc.php

**HTTP Request**: POST

**Parameters**: uid, pwd

**Payload**:

uid = admin' AND (select sleep(10) from dual where database() like 'se%');--

pwd = anythingyouwant

### Code Fix



# Secure Password Storage

Secure password storage involves storing user passwords in a way that protects them from unauthorized access, even in the event of a data breach. They key principles of secure password storage involves hashing and salting. Hashing is a function that takes the password and produces a fixed-sized string of characters. Salting involves generating a unique random value that is appended to the password before hashing.

## Vulnerability Identified: Unsecure Password Storage

Passwords in this web app are stored in plaintext. There is also no salt being used when validating the passwords. Hashing and salts need to be introduced in three locations: at registration, at login, and at password change.

### Code Fix



*In ProcessRegistration*



*In ResetPassword*



*In ProcessLogin*

# Cross-Site Request Forgery

CSRF is a type of exploit that involves an attacker tricking a user’s browser into performing actions on a web application without the user’s knowledge. They typically target web applications vulnerable to manipulation through HTML forms or scripted requests. A malicious actor will target a user that is already authenticated and get them to open web pages that will use the same authenticated cookies as the user in a different tab. These tabs can then execute transactions that appear authentic to the server.

## Vulnerability Identified: CSRF in password reset

There is a CSRF vulnerability when a user changes a password. It is possible for an attacker to use an authenticated user’s session to change their password without their knowledge. The app is also using GET requests to process the password change. The app was changed to use POST requests instead of GET. I have also implemented a CSRF token that is created when the request is first sent and unset after the password has been changed. This will prevent malicious attackers from hijacking an existing session to submit requests and only process requests that were generated by the user.

**Location**: change.php, reset.inc.php

**HTTP Request**: GET

### Code Fix



*In change.php*



*In reset.inc.php*

# Directory Traversal

Directory traversal vulnerabilities allow attackers to access files and directories outside of the intended directory structure of the web application. This issue arises when the web application does not properly sanitize user-supplied input, allowing attackers to manipulate file paths to access the server system files.

## Vulnerability Identified: Directory traversal

The auth2.php page takes an unsanitized parameter to display files. In its current state, it allows an attacker to traverse to higher directories and display files on the server that are not meant to be accessed. I have created a subfolder so that only files that are meant to be read are accessed. I also check the string input for traversal, such as ..\ or ../.

**Location**: auth2.php

**HTTP Request**: GET

**Parameter**: FileToView

**Payloads**:

FileToView=index.php

FileToView=../../readme\_en.txt

### Code Fix



# Brute-Force Attack

Brute-force attacks involves systematically submitting a request to a server to guess password, encryption keys, or authentication credentials. This method relies on the attacker’s ability to automate the process by trying different combinations rapidly. This vulnerability can be mitigated by locking out IP addresses that have submitted multiple failed requests in sequence.

## Vulnerability Identified: Brute-force in registration

The web app contains a vulnerability in the register.php page where a malicious actor can submit multiple requests without being locked out. This vulnerability was mitigated by implementing a lockout check and locking out ip addresses that have multiple failed registrations in sequence.

**Location**: register.php, signup.inc.php

**HTTP Request**: POST

**Behavior**: Submitting multiple requests to register

### Code Fix





*In signup.inc.php*

# Conclusion

This report highlights the importance of identifying and mitigating vulnerabilities in web applications. Significant insights have been gained through the examination of common security threats, such as Cross-Site Scripting, Session Management flaws, SQL Injection, Secure Password Storage deficiencies, Cross-Site Request Forgery, Directory Traversal risks, and Brute-Force Attacks. By addressing each identified vulnerability and implementing mitigation strategies, web developers can fortify their applications against malicious exploitation and unauthorized access. A multi-layered approach to security can be established through the implementation of input validation and sanitization, as well as adopting industry best practices such as parameterized queries, session token management, and IP lockout mechanisms.

Ultimately, embracing a culture of security consciousness, leveraging robust coding practices, and implementing proactive mitigation measures can ensure the confidentiality, integrity, and availability of sensitive data and functionalities for users.