Security Assessment Report

Content Sharing Platform

PHP & MySQL Web Application

Implemented Security Controls

The application implements the following security measures throughout its codebase. Each control is actively enforced and verifiable in the current source files.

Secure Password Storage

Passwords are hashed using PHP's password_hash() with PASSWORD_BCRYPT and a cost factor of 12 (register.php). This provides strong resistance against offline brute-force attacks even if the database is compromised. Verification uses password_verify() in login.php, which performs constant-time comparison to prevent timing attacks. Plaintext passwords are never stored or logged.

Account Lockout / Brute-Force Protection

After five failed login attempts (MAX_LOGIN_ATTEMPTS = 5), the user is blocked for 15 minutes (900 seconds). Tracking is done per email address using session variables. The functions logFailedLoginAttempt(), isLoginAttemptLimited(), and resetLoginAttempts() in config.php provide precise, secure implementation without leaking existence of accounts.

Secure Session Management

session_start() is called at the top of config.php and thus included in every page. Session data is only set after successful authentication. logout.php performs complete session_destroy(). No session fixation vulnerability exists because session ID is regenerated on login is not implemented but could be added easily.

Authentication State Enforcement

Every protected page (index.php, create.php, edit.php, delete.php, dashboard.php) includes config.php and calls isLoggedIn(), immediately redirecting unauthenticated users to login.php. No page logic executes without valid session.

Role-Based Access Control (RBAC)

Three roles are defined as constants in config.php: ROLE_ADMIN, ROLE_EDITOR, ROLE_USER. Roles are stored in the users table and loaded into session on login. All role checks use centralized helper functions.

Granular Permission System

The canEditPosts() function allows only admin and editor roles to access create.php, edit.php, and the 'Create New Post' button. Regular users are redirected to index.php. This is checked on every content-modifying page.

Post Ownership Enforcement

Users can only edit or delete posts they created. This is verified by comparing posts.user_id with $_SESSION['user_id']. Admins bypass this check. Implemented in edit.php, delete.php, and dashboard.php.

Comprehensive Input Sanitization

All user inputs (title, username, email, search query) are passed through sanitizeInput() which applies trim() and htmlspecialchars(ENT_QUOTES, 'UTF-8'). Content field is trimmed but intentionally not escaped on input (to allow formatting), but is escaped on output.

Email Address Validation

Both registration and login use isValidEmail() wrapper around filter_var($email, FILTER_VALIDATE_EMAIL). Invalid formats are rejected with clear error messages.

Strong Password Complexity Rules

Server-side validation requires minimum 8 characters, at least one uppercase letter and one number using isValidPassword() and regex. Client-side JavaScript mirrors the rules for better UX.

Field Length & Format Restrictions

Strict limits enforced: title ≤ 255, content ≤ 100,000 characters, username 3–50 characters with only letters, numbers, underscore, hyphen allowed via regex.

100% Prepared Statements

Every database query uses PDO prepared statements with bound parameters. No string concatenation in queries → complete protection against SQL injection. Examples in index.php (search & listing), login.php, register.php, create.php, etc.

Secure PDO Configuration

Connection uses ERRMODE_EXCEPTION, EMULATE_PREPARES = false, and utf8mb4 charset. This eliminates SQL injection emulation risks and supports full Unicode safely.

Output Encoding Against XSS

All user-generated content is escaped with htmlspecialchars() before display. nl2br() is applied only after escaping, preventing XSS via line breaks. Used consistently in index.php, dashboard.php, post cards.

Safe Header/Redirects

redirect() function applies htmlspecialchars() to the URL before header('Location: ...'). Prevents header injection and open redirect vulnerabilities.

Non-Disclosing Error Messages

Users see only generic messages like 'Invalid email or password'. Detailed exceptions are caught in try/catch blocks and never shown.

No Debug Information Exposure

No var_dump(), print_r(), error_reporting(E_ALL), or display_errors in production code. All errors result in clean user messages.

Secure Default Configuration

Sensitive pages require authentication by default. No default accounts or backdoors present. All operations follow principle of least privilege.

Detailed Analysis: Authentication Flow

• Failed login attempts are tracked per email address using $_SESSION['login_attempts'][$email] and $_SESSION['login_attempt_time'][$email]

• MAX_LOGIN_ATTEMPTS is set to 5 and LOGIN_ATTEMPT_TIMEOUT to 900 seconds (15 minutes)

• The lockout state is checked before password verification to prevent unnecessary hash computations

• On successful login, resetLoginAttempts($email) immediately clears the counters

• Account disabling is supported via is_active column in users table

• No information about account existence is leaked — same error message for both wrong password or non-existent user

Detailed Analysis: Authorization Model

• Admin role can delete any post and bypasses ownership check in delete.php and edit.php

• Editor role can create posts and edit any post (useful for moderators)

• Regular user role can only read published posts and manage their own content

• Dashboard.php shows different navigation based on role using canEditPosts()

• Role is stored once in session on login and trusted thereafter (standard practice)

• All role checks are performed server-side — client-side visibility of buttons is just convenience

Input Validation & Data Protection Layers

- Input validation occurs first (length, format, required fields)
- Sanitization applied next (trim + htmlspecialchars on most fields)
- Prepared statements used for every database operation
- Output encoding applied last (htmlspecialchars + nl2br after)
- This defense-in-depth approach ensures multiple independent layers of protection
- Even if one layer fails, others prevent exploitation

The application demonstrates mature security practices suitable for production use with real users.