ISUPPLY CYBERSECURITY HACKATHON TECHNICAL REPORT

Submitted by: Youssef Sherif Hassan

Email: yosherif15@gmail.com

Table of Contents

1. Executive Summary

This report presents a security audit of a Laravel-based application deployed within a Docker environment. The goal was to identify vulnerabilities, simulate potential attacks, and provide recommended remediations. Key vulnerabilities were found in both the application code and Docker configurations.

2. Project Introduction

This audit was conducted as part of the iSUPPLY Cybersecurity Hackathon. The objective was to assess the security posture of a Laravel application running inside Docker. We utilized static analysis, configuration review, and vulnerability simulation techniques to achieve our goals.

3. Methodology

We began with static code analysis to identify common vulnerabilities such as SQL Injection and XSS. This was followed by a thorough inspection of Dockerfiles and Compose files to identify misconfigurations. The OWASP Top 10 and SANS 25 frameworks guided our vulnerability classification. Finally, simulations of real-world attack scenarios were performed, and fixes were proposed or implemented.

4. Docker Vulnerability Findings

4.1 Title: Directory Listing Enabled (Autoindex)

Description: The Nginx configuration file has autoindex enabled for the /storage/app/public/ location block.

Impact: High

Location: docker/docker.conf

Suggested Fix: Set autoindex off; in Nginx configuration.

4.2 Title: HTTP Only (No HTTPS)

Description: The server listens only on HTTP (port 80) without redirecting or enforcing HTTPS.

Impact: Medium

Location: docker/default.conf

Suggested Fix: In production, enforce HTTPS with SSL certificates and redirect HTTP to HTTPS.

4.3 Title: Access to Hidden Files (Dotfiles) Not Restricted

Description: Access to files like .env and .git is not restricted in the Nginx config.

Impact: Medium

Location: docker/default.conf

Suggested Fix: Uncomment or add rules to deny access to dotfiles in the Nginx configuration.

4.4 Title: No Request Size or Rate Limiting

Description: Lack of limits on client request size and frequency can lead to DoS attacks.

Impact: Low

Location: docker/default.conf

Suggested Fix: Add security measures to limit both the size of incoming requests and request frequency.

4.5 Title: Running as Root by Default

Description: Most services run using the root user, increasing container escape risk.

Impact: High

Location: Dockerfile

Suggested Fix: Use a dedicated non-root user for running services.

4.6 Title: Hardcoded Sensitive Credentials

Description: Secrets are hardcoded in the image via the .env file.

Impact: High

Location: Dockerfile

Suggested Fix: Use environment variables or secrets managers instead of hardcoded values.

4.7 Title: Application Debug Mode Enabled

Description: APP_DEBUG=true and APP_ENV=local are set for the Laravel application.

Impact: Medium

Location: Dockerfile

Suggested Fix: Set APP_DEBUG=false and APP_ENV=production in production environments.

4.8 Title: File and Folder Permissions Too Open

Description: chmod 777 gives full access to all users inside the container.

Impact: Medium

Location: Dockerfile

Suggested Fix: Use least privilege permissions like 755 or 644.

4.9 Title: Unverified External Sources

Description: Scripts and packages are downloaded without verifying authenticity.

Impact: Medium

Location: Dockerfile

Suggested Fix: Use cryptographic checksums or trusted certificates to validate sources.

5.0 Title: Verbose PHP Error Display Enabled

Description: PHP is configured to display errors publicly (display_errors = On).

Impact: High

Location: docker/php.ini

Suggested Fix: Disable public error output; enable logging only.

5.1 Title: File Uploads and URL File Access Enabled Without Logging

Description: Logging is disabled despite file_uploads and allow_url_fopen being enabled.

Impact: Medium

Location: docker/php.ini

Suggested Fix: Enable log_errors or disable unused PHP features.

5.2 Title: Empty MySQL Passwords Allowed

Description: MYSQL_ALLOW_EMPTY_PASSWORD=1 is set.

Impact: High

Location: docker-compose.yml → environment section

Suggested Fix: Remove this or set it to 0 and define a strong root password.

5.3 Title: Using MySQL Root User

Description: The application uses the root user for MySQL access.

Impact: High

Location: docker-compose.yml → environment section

Suggested Fix: Create a limited MySQL user for application access.

5.4 Title: Port Exposure

Description: Exposing ports 80 and 8080 can increase the attack surface.

Impact: Medium

Location: docker-compose.yml → ports section

Suggested Fix: Restrict exposed ports in production using firewalls or reverse proxies.

5.5 Title: Insecure Volume Mount

Description: Mounting .:/var/www/html as read-write allows host-based file tampering.

Impact: Medium

Location: docker-compose.yml → volumes section

Suggested Fix: Use a read-only mount or build Docker images with the source code pre-bundled.

Laravel’s Vulnerabilities

5.6 [Hardcoded Admin Credentials in Seeder]

- **Description:**
  The `DatabaseSeeder` manually creates an admin user with a hardcoded email address (`test@example.com`) and no password specified.
- **Impact:**
  Medium – If this seeder is executed in production without overriding the password, it could create a default admin account with a predictable identity and possibly an insecure default password.
- **Location:**
  `database/seeders/DatabaseSeeder.php`
- **Suggested Fix:**
  Do not hardcode admin accounts or credentials. Use environment variables or secure secrets when seeding sensitive users, and avoid including seeders like this in production builds.

5.7[Insecure Default Password in User Factory]

- **Description:**
  The `UserFactory` uses a hardcoded, weak default password (`12345`) that is applied to all generated users. Even though it's in a seeder context, such weak credentials pose a risk if leaked or used beyond test environments.
- **Impact:**
  High—Predictable passwords for user accounts could be exploited if they make it into production (e.g., for test or admin users).
- **Location:**
  `database/factories/UserFactory.php`
- **Suggested Fix:**
  Use hashed and randomized passwords for each test user. Ensure the password is securely generated using Laravel’s `Hash::make()` and never hardcoded in shared code.

5.8 [Deeply Nested Fake HTML Content in Post Factory]

- **Description:** The `PostFactory` generates random HTML content using `randomHtml(maxDepth: 20)`, which produces deeply nested HTML.
- **Impact:**
  Medium – May lead to performance issues, rendering problems, or XSS-like behavior in views if the output is not sanitized when displayed.

- **Location:**
  `database/factories/PostFactory.php`
- **Suggested Fix:**
  Limit HTML nesting depth or use plain text or Markdown instead. Always sanitize and escape content before displaying it in the frontend.

5.9 [Public Access to Sensitive API Route]

- **Description:**
  The `/users` route is meant to be **protected by authentication middleware**, but the line enabling it is commented out. As a result, anyone — including unauthenticated users — can access this endpoint and potentially retrieve sensitive user data via `ListUsersController`.
- **Impact:**
  High – If `ListUsersController` exposes user information (names, emails, etc.), it could result in a data leak, user enumeration, or even account compromise if combined with other issues.
- **Location:**
  `routes/api.php`
- **Suggested Fix:**
  Uncomment the route group and protect the `/users` endpoint with `auth:sanctum` (or other authentication middleware). Ensure all API endpoints that expose user or system data require authentication and authorization.

6.0 [Registration Route Enabled in Production]

- **Description:**
  The app exposes a public registration page (`/register`) that allows **anyone** to create a user account.
- **Impact:**
  Medium – In production environments, this can allow bots or malicious actors to spam user creation or escalate privileges if role assignments are not validated properly.
- **Location:**
  `routes/web.php` → Inside `guest` group
- **Suggested Fix:**
  Disable or restrict registration in production using a feature flag or environment check. You can conditionally register these routes only in development.

6.1 [No CAPTCHA or Rate Limiting on Login/Register]

- **Description:**
  Login, registration, and password reset routes (`POST /login`, `/register`, `/forgot-password`) are not visibly protected by throttling or CAPTCHA.
- **Impact:**
  High – These endpoints are susceptible to brute-force attacks, user enumeration, or abuse.
- **Location:**
  `routes/web.php`
- **Suggested Fix:**
  Use Laravel's built-in rate limiting middleware (like `throttle:10,1`) and integrate CAPTCHA (like Google reCAPTCHA) on login/register forms to reduce abuse.

6.2 [Password Reset Routes Public Without Rate Limiting]

- **Description:**
  Password reset requests (`/forgot-password`) are public and do not appear to have throttling, which may allow mass email spamming or token generation attacks.
- **Impact:**
  Medium—Attackers can spam reset requests or harvest user existence via responses.
- **Location:**
  `routes/web.php` → Password reset section
- **Suggested Fix:**
  Apply rate limiting (`throttle`) middleware to these routes and ensure responses are generic (e.g., "If your email exists…").

6.3 [Welcome Page Leaks System Versions]

- **Description:**
  The / route (homepage) returns `laravelVersion` and `phpVersion` to the frontent
- **Impact:**
  Medium – This can help attackers craft version-specific exploits (e.g., known vulnerabilities in Laravel X.X or PHP Y.Y).
- **Location:**
  `routes/web.php`
- **Suggested Fix:**
  Remove version data from the response in production. Display only in development environments.

6.4 [Posts Resource Fully Exposed to Authenticated Users]

- **Description:**
  The `Route::resource('posts', PostsController::class);` exposes all RESTful endpoints for blog posts (index, create, store, show, edit, update, destroy), but there are **no additional access controls** like roles or ownership checks.
- **Impact:**
  Medium – Any authenticated user may be able to view, edit, or delete posts they don't own (depends on controller logic).
- **Location:**
  `routes/web.php`
- **Suggested Fix:**
  Apply authorization logic in the `PostsController` (e.g., using Laravel's policies or gates) to ensure users can only modify their own posts.

6.5 [Insecure File Upload Naming]

- **Description:**
  The `storeFile()` method saves uploaded files using their **original client-provided name**:

```
$file->getClientOriginalName()
```

- **Impact:**
  Medium – Could allow attackers to overwrite existing files or predict filenames for unauthorized access.
- **Location:**
  `app/Http/Controllers/Controller.php`
- **Suggested Fix:**
  Sanitize and randomize filenames before storing them. Use Laravel's built-in method.

6.6 [Flawed Admin Check Logic]

- **Description:**
  The controller tries to check if the user is an admin:

However, this logic is **inverted**:

  - If the user **is not** an admin, `$is_admin` becomes `true`
  - The variable is named `isAdmin` but its meaning is opposite

- **Impact:**
  Medium – Can cause incorrect permission logic in the frontend (e.g., hiding admin features from actual admins and showing them to regular users)
- **Location:**
  `app/Http/Controllers/DashboardController.php`
- **Suggested Fix:**
  Correct the logic and naming.

6.7 [Mass Assignment Risk in Profile Update]

- **Description:**
  The method responsible for updating user profile information applies all received user input directly to the user model without explicitly selecting which fields should be updated. This assumes that only safe and expected fields are present, relying on Laravel's internal field protection mechanism.
- **Impact:**
  Medium – If the application's underlying model configuration (`$fillable`) is not strictly defined, this could allow attackers or misconfigured clients to update unintended attributes. For example, fields related to user roles, admin privileges, or internal flags could be manipulated if not properly restricted.
- **Location:**
  `ProfileController` → `update()` method
- **Suggested Fix:**
  Limit updates to only specific, intended fields (e.g., name and email). This reduces the risk of overwriting sensitive or system-level attributes. The team should audit the user model and ensure only safe fields are allowed for mass assignment, and update the logic to reference only those fields explicitly.

6.8 [No Access Control on Sensitive User Search]

- **Description:**
  The controller allows any request (even unauthenticated ones) to search and retrieve user data—including sensitive attributes such as names, emails, and national IDs. There's **no authentication or authorization** applied to this endpoint, which can lead to **data exposure or user enumeration** attacks.
- **Impact:**
  High – Without access control, attackers could harvest or enumerate all users in the system, including private identifiers.
- **Location:**
  `ListUsersController` → `__invoke()` method

(Also relevant to `routes/api.php`, where this route is currently **not protected** by middleware)

- **Suggested Fix:**
  Restrict this endpoint using authentication middleware (such as token-based or session-based guards) and implement **role-based authorization** to ensure only authorized users (e.g., admins) can access user search.

6.9 [Exposing Personally Identifiable Information (PII)]

- **Description:**
  The search function returns potentially sensitive personal information (name, email, national ID) without any masking or access restrictions. While wrapped in a resource (`UserSearchResource`), there's no indication that fields are redacted or obfuscated.
- **Impact:**
  High – Public or unauthorized access to national IDs and emails can lead to identity theft, phishing, or privacy violations.
- 4\23Location: `ListUsersController.php file`
- **Suggested Fix:**
  Limit the data fields returned by `UserSearchResource`, especially for sensitive identifiers. Consider returning only non-sensitive information unless access is explicitly authorized.

7.0 [No Rate Limiting or Abuse Protection]

- **Description:**
  There's no evidence of rate limiting or throttling on this search endpoint, meaning an attacker could brute-force requests or crawl the user base rapidly.
- **Impact:**
  Medium – Could enable mass enumeration or cause performance issues.
- Location: ListUsersController.php file
- **Suggested Fix:**
  Apply rate limiting using Laravel's API throttling middleware and consider adding CAPTCHA or abuse detection for UI-based calls.

7.1 [Password Stored Without Hashing]

Description:
In the user registration logic, the password is stored directly using `$request->password` without applying hashing. This results in plain-text passwords being saved to the database, making the application critically vulnerable if the database is ever compromised.

Impact:
High – Plain-text passwords expose all users to credential theft and reuse attacks. This violates basic security hygiene and puts all accounts at severe risk.

Location:
`RegisteredUserController.php → store()` method

Suggested Fix:
Before saving, hash the password using Laravel's `Hash::make()` method:

7.2 [User Object Exposed Globally to Frontend]

Description:
In the `share()` method, the authenticated user object (`$request->user()`) is made globally available to the frontend through the `auth` prop. If not carefully filtered, this can expose sensitive user attributes (e.g., tokens, admin flags, or PII) to the client-side application.

Impact:
Medium – If sensitive user data is unintentionally exposed to JavaScript, it may be accessible via browser dev tools or manipulated by attackers using XSS.

Location:
security-task\app\Http\Middleware `HandleInertiaRequests.php → share()`

Suggested Fix:
Limit the shared user data to only the necessary attributes.

7.3 [SQL Injection & Insecure Authentication in Login Logic]

Description:
The `authenticate()` method performs raw SQL using unsanitized user input for the email field:

```
$user = DB::select("SELECT * FROM users WHERE email = '" . $email . "'
LIMIT 1");
```

It also directly compares the plaintext input password with the stored password from the database, assuming passwords are stored unhashed. Additionally, Laravel's secure `Auth::attempt()` method is commented out and bypassed.

Impact:
High – This introduces:

- **SQL Injection risk**, since user input is directly embedded into the SQL query.
- **Authentication bypass potential**, especially if passwords are not hashed.
- **Exposure of plaintext passwords** in logs via `logger()` calls.
- **Bypassing Laravel's rate limiting and lockout system**, since `ensureIsNotRateLimited()` is commented out.

Location:
`LoginRequest.php → authenticate()`

Suggested Fix:

- Use parameterized queries or Laravel's Eloquent model to fetch users.
- Do not log plaintext passwords.
- Use Laravel's built-in `Auth::attempt()` or `Hash::check()` for password verification.
- Uncomment and enforce rate limiting via `ensureIsNotRateLimited()` for brute-force protection.
- Ensure passwords are hashed in the database using Laravel's `Hash::make()`.

7.4 [Unvalidated File Access via Public Storage]

Description:
The model uses `Storage::disk('public')->url($this->attachment_path)` to generate file URLs for post attachments. If `attachment_path` is not strictly validated or sanitized, users may access unintended files stored in the public disk, potentially leading to **information disclosure**.

Impact:
Medium – If user-uploaded file paths are not securely validated, this could allow access to arbitrary files in the public storage, including internal documents, logs, or other users' data.

Location:
`Post.php → attachmentUrl() method`

Suggested Fix:

- Ensure uploaded files are stored in a strict, isolated directory (e.g., `uploads/posts/`).
- Sanitize and validate the `attachment_path` before storing.
- Limit the file types and extensions allowed.

7.5 [Mass Assignment Risk via Unrestricted $guarded]

Description:
The `User` model uses `$guarded = []`, which means **no attributes are protected from mass assignment**. This allows any user-controlled data to be written to the model, including sensitive fields like `user_type`, `id`, or other internal flags if not properly filtered in the controller/request logic.

Impact:
High – If a developer accidentally uses `$request->all()` or `fill()` without proper filtering, an attacker could escalate privileges (e.g., becoming an admin) or corrupt user records.

Location:
`User.php` model – `$guarded = []`

Suggested Fix:

- Replace `$guarded = []` with an explicit `$fillable` array that **only includes safe fields** (e.g., `name`, `email`, `password`, etc.).
- Review all form requests and controllers for use of `fill()`, `create()`, or `update()` with unfiltered data.
- Ensure no sensitive fields (e.g., `user_type`, `is_admin`, `wallet_balance`) can be manipulated by the user.

7.6 [Over-Permissive Authorization Logic in Post Policy]

Description:
The `PostPolicy` grants **unconditional access** (`return true`) to all users for sensitive actions such as `view`, `create`, and `update`. This means **any authenticated user can view and modify any post**, regardless of ownership or privilege level. It bypasses expected access control boundaries.

Impact:
Medium to High – May lead to **unauthorized post creation, viewing, or updates**, which can result in **data leakage, tampering, or abuse**.

Location:
`PostPolicy.php` – methods `viewAny()`, `view()`, `create()`, and `update()`

Suggested Fix:

- Enforce **role-based** or **ownership-based** logic:
  - Allow users to view/update **only their own posts**, unless they are Admin.

    - Use checks like `$user->id === $post->author_id` or `$user->user_type === UserType::Admin->value`
- Avoid returning `true` unconditionally for sensitive methods

7.7 [Missing Policy Registration for Access Control]

Description:
Although a `PostPolicy` exists and is correctly defined in the application, **it is not explicitly registered** in the `AppServiceProvider` or a dedicated `AuthServiceProvider`. Laravel typically requires policies to be registered to enforce model-level authorization automatically.

Impact:
Medium – If policies are not properly registered, **authorization logic may not be triggered**, potentially allowing unauthorized actions (e.g., post edits, deletions) to proceed without checks.

Location:
`AppServiceProvider.php` – No policy registration found
Suggested Fix:

- Ensure that `PostPolicy` is registered in `AuthServiceProvider.php`
- Verify that `Gate::authorize()` calls rely on the correct policy mappings.

5. Docker Security Review

The Dockerfile used in the project was found to run as root and lacked security hardening. We recommend using a non-root user, minimizing image layers, and performing vulnerability scanning using tools such as Trivy.

7. Lessons Learned

This project deepened our understanding of Laravel and Docker security practices. We learned how to identify and mitigate real-world vulnerabilities and gained experience in secure coding and DevSecOps.

8. References

- OWASP Top 10
- Laravel Security Best Practices
- Docker Hardening Guide

9. Appendix

Docker Configuration Vulnerabilities Table

#	Vulnerability Title	Problem Description	Location	Fix Applied
1	Directory Listing Enabled	Nginx autoindex enabled in /storage/app/public.	docker/docker.conf	Fixed
2	HTTP Only (No HTTPS)	Server only accepts HTTP, no HTTPS enforcement.	docker/default.conf	Fixed
3	Access to Dotfiles	Access to hidden files (.env, .git) not restricted.	docker/default.conf	Fixed
4	No Request Size or Rate Limiting	No controls on request size/frequency.	docker/default.conf	Fixed
5	Running as Root	All services run as root user.	Dockerfile	Fixed
6	Hardcoded Credentials	Sensitive info in image via .env file.	Dockerfile	Fixed
7	Debug Mode Enabled	APP_DEBUG=true and APP_ENV=local in Dockerfile.	Dockerfile	Fixed
8	Over-Permissive Permissions	chmod 777 used in Dockerfile.	Dockerfile	Fixed
9	Unverified External Sources	Scripts fetched without checksum validation.	Dockerfile	Fixed
10	Verbose PHP Error Display	display_errors = On in PHP config.	docker/php.ini	Fixed
11	File Upload & URL Access Unlogged	Logging disabled with file_uploads & allow_url_fopen enabled.	docker/php.ini	Fixed
12	Empty MySQL Passwords	MYSQL_ALLOW_EMPTY_PASSWORD=1 set.	docker-compose.yml	Fixed
13	Using MySQL Root User	Root used for DB access.	docker-compose.yml	Fixed
14	Port Exposure	Ports 80, 8080 are exposed publicly.	docker-compose.yml	Fixed
15	Insecure	Mounted .:/var/www/html as read-	docker-	Fixed

	Volume Mount	write.	compose.yml	

Laravel Configuration Vulnerabilities Table

#	Vulnerability Title	Problem Description	Location	Fix Applied
1	Hardcoded Admin Credentials	Admin created with hardcoded credentials.	database/seeders/DatabaseSeeder.php	Fixed
2	Weak Default Password in Factory	All users get '12345' as default password.	database/factories/UserFactory.php	Fixed
3	Public Access to /users API	Route not protected by auth middleware.	routes/api.php	Fixed
4	Public Registration in Prod	Accessible /register in production.	routes/web.php	Fixed
5	No CAPTCHA or Rate Limiting	Missing protections on auth routes.	routes/web.php	Fixed
6	Password Reset Route Unprotected	No rate limiting on /forgot-password.	routes/web.php	Fixed
7	Version Info Leak	Homepage exposes Laravel & PHP versions.	routes/web.php	Fixed
8	Post Resource Unrestricted	All authenticated users can access post actions.	routes/web.php / PostsController	Fixed
9	Insecure File Upload Naming	Uses client filenames without sanitization.	app/Http/Controllers/Controller.php	Fixed
10	Flawed Admin Check Logic	Inverted logic in admin check.	DashboardController.php	Fixed
11	Mass Assignment in Profile	Applies all user input blindly.	ProfileController → update()	Fixed
12	User Search Unrestricted	Allows access to sensitive info without auth.	ListUsersController.php	Fixed
13	PII Exposed in	Returns national	UserSearchResource	Fixed

	Search	IDs & emails unmasked.		
14	No Abuse Protection on Search	Lack of throttling/CAPTCHA on search endpoint.	ListUsersController.php	Fixed
15	Plaintext Password Storage	Password stored without hashing.	RegisteredUserController.php	Fixed
16	Frontend Exposes Full User Object	$request->user() shared globally.	HandleInertiaRequests.php	Fixed
17	SQL Injection in Login	Raw SQL with no sanitization in login logic.	LoginRequest.php	Fixed
18	Unvalidated Public Storage URLs	File path access not validated.	Post.php	Fixed
19	Mass Assignment via $guarded = []	No protection against mass assignment.	User.php	Fixed
20	PostPolicy Grants Full Access	All users allowed to create/edit posts.	PostPolicy.php	Fixed
21	Policy Not Registered	PostPolicy not linked to model.	AppServiceProvider.php	Fixed