



Security Audit of Putimg

Application Security 2024/2025

Rosario Pavone ER1799
Edgar Benito ER1791

Table of Contents

1. Black-box Approach	3
1.1 Manual basic Vulnerabilities	3
1.2 Nikto Assessment.....	4
1.3 OWASP ZAP Findings	5
1.4 Capturing cookies analysis.....	7
1.5 SQLMap Analysis.....	8
1.6 Dirb Directory Enumeration	10
1.7 Curl Request Analysis.....	10
1.8 Attempt to Upload Malicious Files	11
1.9 Recommendations and Conclusion	13
2. White Box	14
2.1 Laravel Controllers	14
2.1.1 ImageController	14
2.1.2 ProfileController	15
Database Problem:	17
2.1.3 UploadController	18
2.2 Middleware	24
2.2.1 EnsureEmailIsVerified.php.....	24
2.2.2 RedirectIfNotVerified.php	25
2.3 Livewire Files	26
2.3.1 LoginForm.php	26
2.3.2 Logout.php	28
2.3.3 Summary Table of Issues and Mitigation Strategies	29

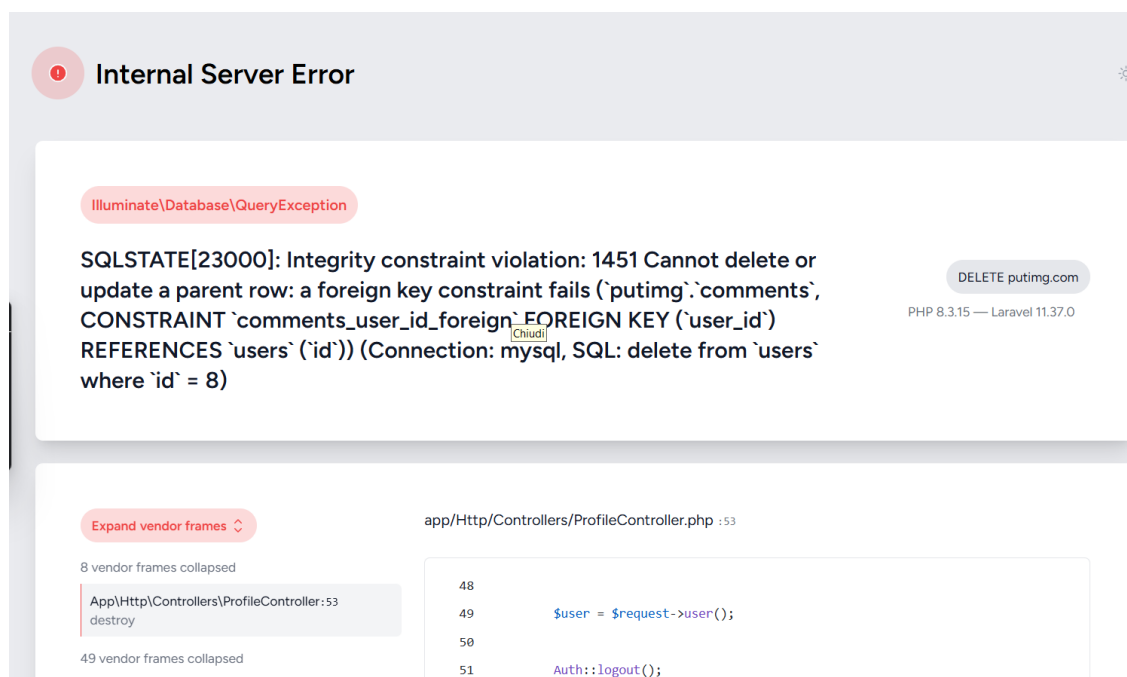
2.4 Models.....	30
2.4.1 Comment.php.....	30
2.4.2 CommentLike.php.....	31
2.4.3 Image.php.....	32
2.5 Requests	34
2.5.1 CustomEmailVerificationRequest.php	34
2.5.2 ProfileUpdateRequest.php	35
2.6 Kernel.php.....	37
2.6.1 Summary Table	39
3. Recommendations and Conclusion.....	40

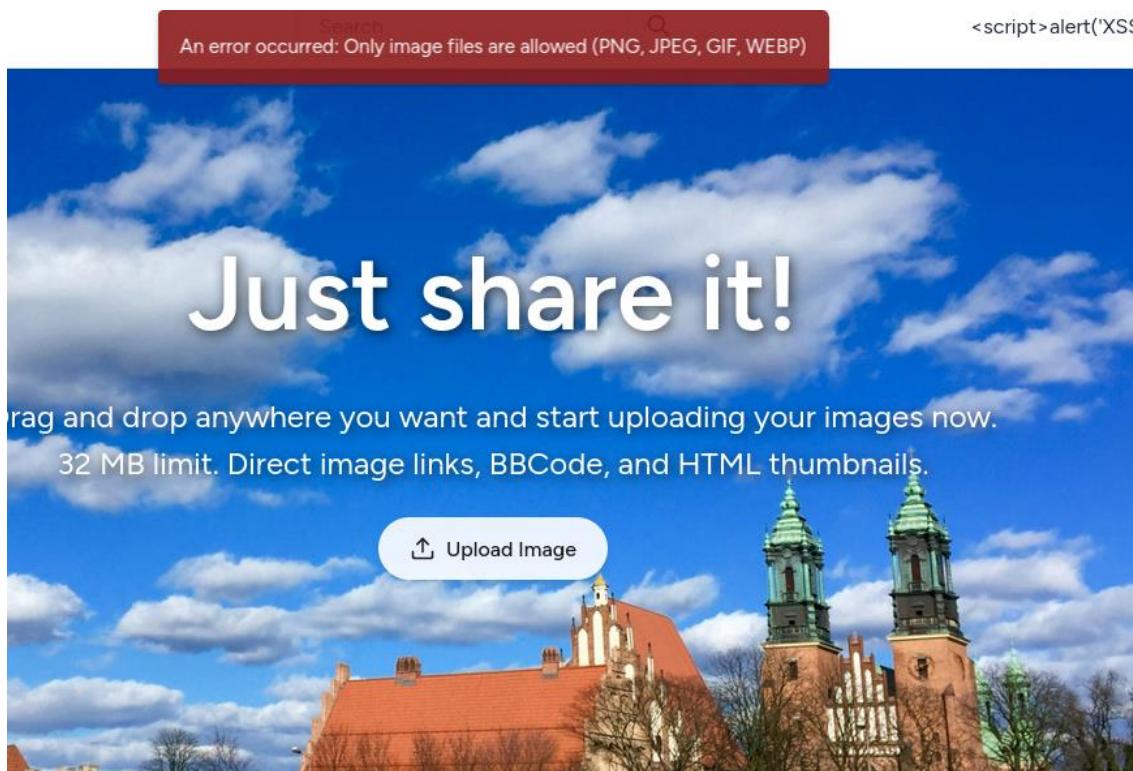
1. Black-box Approach

1.1 Manual basic Vulnerabilities

A black-box testing approach was utilized to identify potential vulnerabilities in the application. During the exploratory analysis, several issues were identified.

One critical problem noted was the malfunction of the "delete account" functionality, which does not execute as expected. Additionally, the upload functionality encountered problems when handling large files; the process would hang indefinitely, failing without providing feedback to the user. These issues indicate insufficient error handling and a lack of robust validation mechanisms.





To address these issues, it is recommended to implement proper error-handling routines and file size validation on both the client and server sides. Users should receive clear feedback if their actions fail, ensuring a smoother experience and mitigating potential abuse of the upload feature.

1.2 Nikto Assessment

Nikto was employed to analyze the server at <https://putimg.com/>. The tool scanned the server for misconfigurations and potential security vulnerabilities using the command:

```
nikto -h https://putimg.com/
```

The scan produced several findings:

Strict-Transport-Security (HSTS) Not Defined: The absence of the HSTS header makes the server vulnerable to SSL stripping attacks, where an attacker forces the user to connect via HTTP instead of HTTPS. This can expose sensitive data during transmission.

Lack of the X-Content-Type-Options Header: This allows browsers to infer MIME types, potentially enabling malicious scripts to execute.

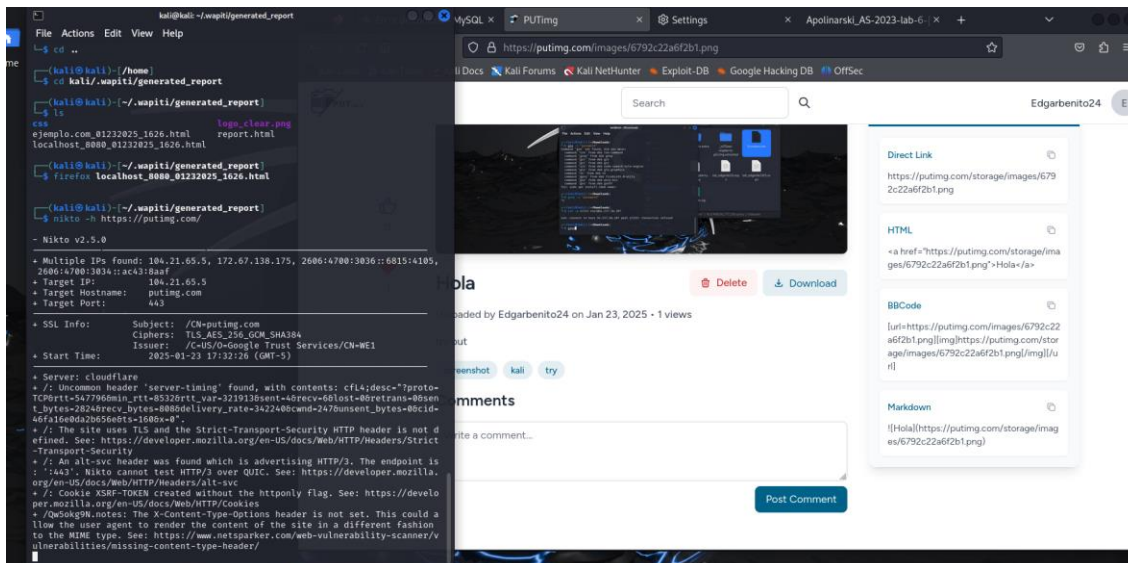
Cookies Without the HttpOnly Flag: The `XSRF-TOKEN` cookie lacks the `HttpOnly` attribute, leaving it accessible to JavaScript and increasing the risk of XSS attacks.

To remediate these issues, the server should:

Configure the HSTS header to enforce HTTPS connections by adding `Strict-Transport-Security: max-age=31536000; includeSubDomains`.

Add X-Content-Type-Options: nosniff to prevent MIME type inference.

Mark sensitive cookies with the HttpOnly and Secure flags to limit client-side access and enforce secure transmission.



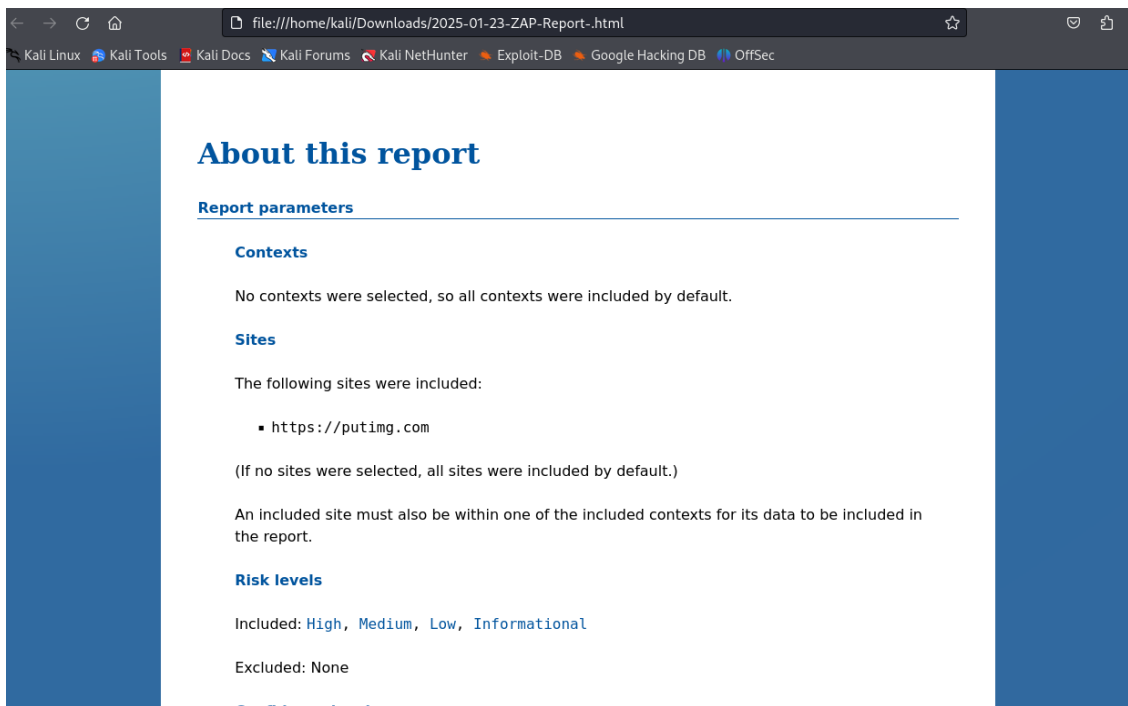
1.3 OWASP ZAP Findings

OWASP ZAP was utilized to perform an active scan and identify security vulnerabilities in the application. The scan revealed several critical issues:

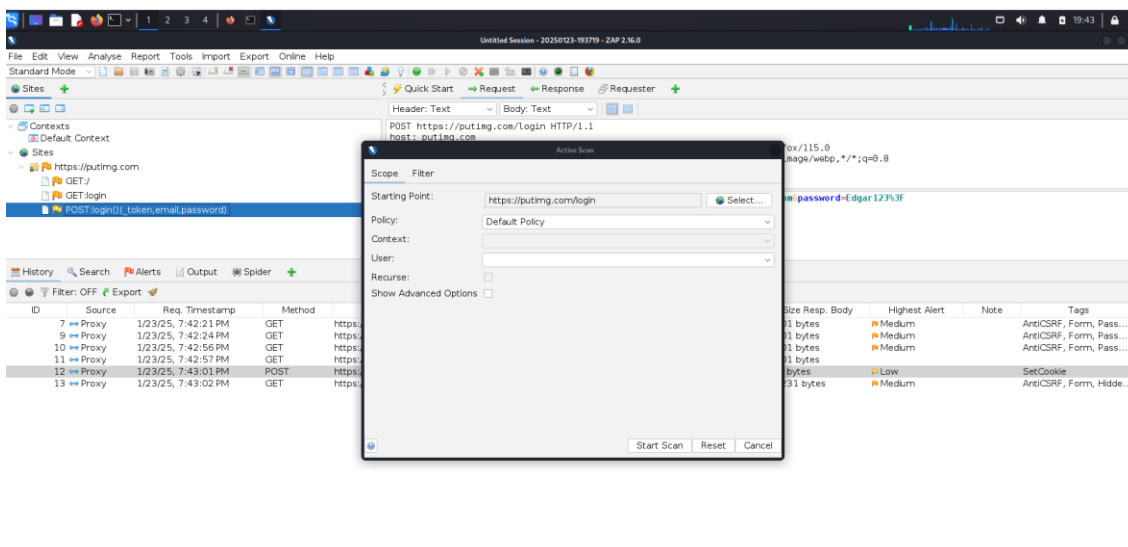
Content Security Policy (CSP) Header Missing: This exposes the application to XSS and other injection attacks, as unauthorized scripts could be executed on the client side.

Sensitive Cookies Without HttpOnly: Similar to the Nikto findings, cookies were accessible to JavaScript, posing a risk if an XSS vulnerability is exploited.

Cross-Domain JavaScript Inclusion: JavaScript files from external sources are included without adequate validation, which could lead to the execution of malicious scripts.



We also use ZAP as a proxy, as a man in the middle, and we obtain in the POST of the Login the user credentials that performs this login.



Recommendations include:

Implement a CSP header to define trusted content sources, for example: Content-Security-Policy: default-src 'self'; script-src 'self' 'trusted-cdn.com';

Configure all sensitive cookies with the HttpOnly and Secure flags.

Audit and whitelist external JavaScript sources to ensure their reliability.

1.4 Capturing cookies analysis

Cookies were captured during testing using browser developer tools, specifically to examine how session handling is implemented within the application. By intercepting the cookies sent during user interactions, it was possible to identify weaknesses in their configuration.

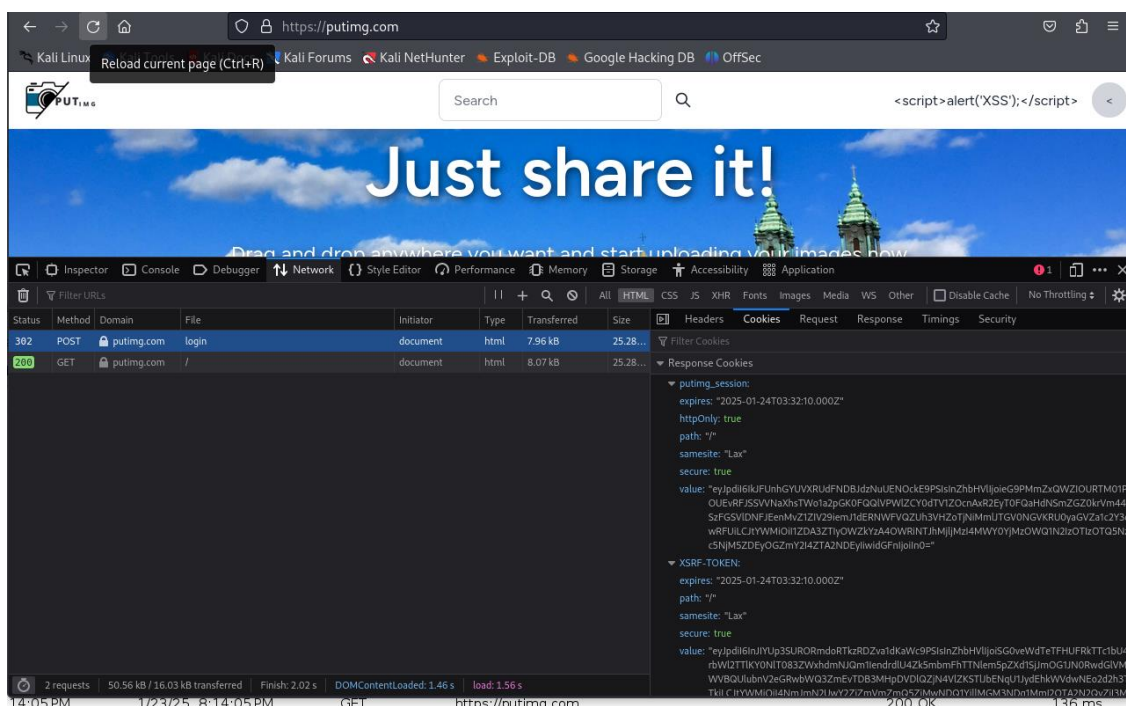
One key observation was the absence of critical security attributes such as `HttpOnly` and `Secure` on some session cookies. This configuration allows JavaScript running in the browser to access these cookies, increasing the risk of session hijacking if an attacker successfully executes a Cross-Site Scripting (XSS) attack.

To mitigate these risks and strengthen cookie security, the following measures are recommended:

Enable the `HttpOnly` Attribute: This prevents JavaScript from accessing cookies, ensuring that session data is protected from XSS attacks.

Enable the `Secure` Attribute: This ensures that cookies are transmitted only over encrypted HTTPS connections, reducing the risk of interception during transit.

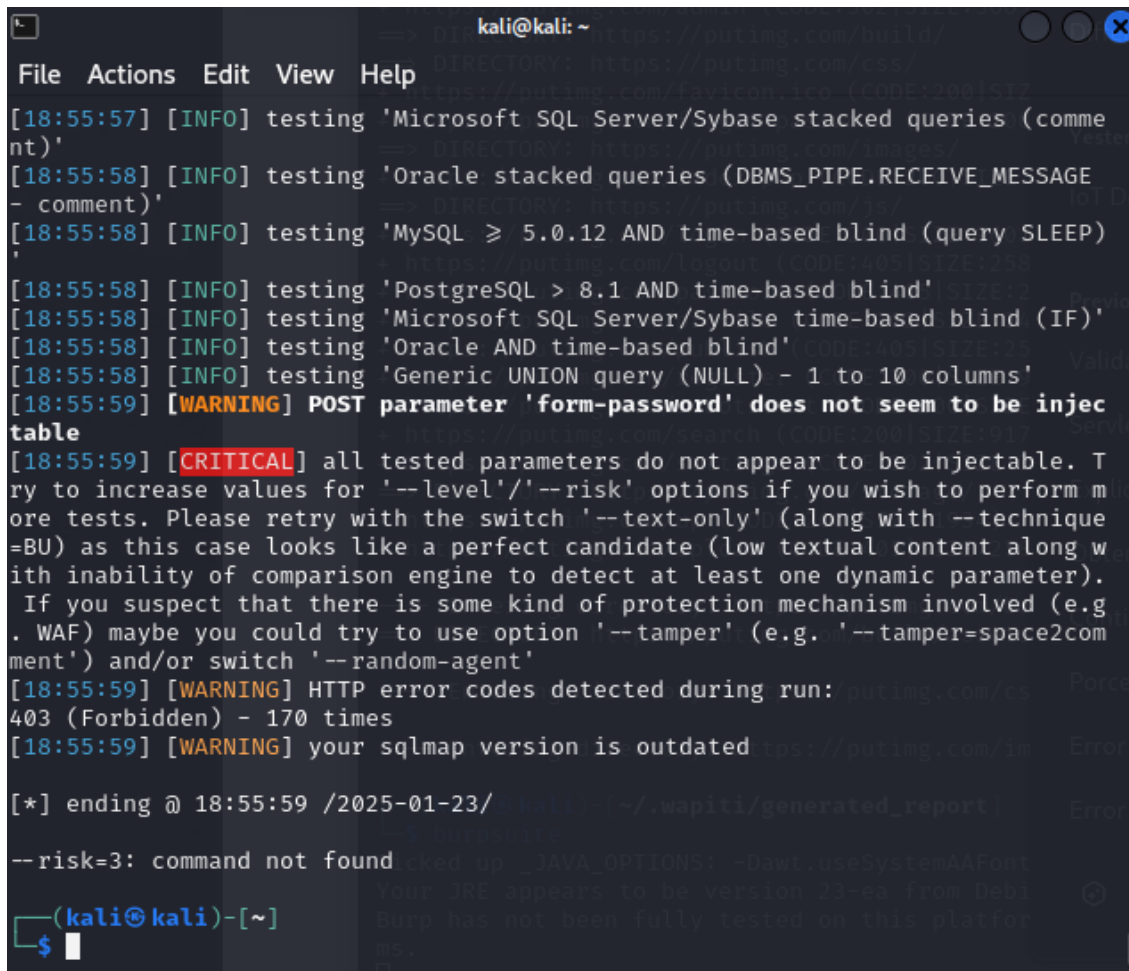
Set the `SameSite` Attribute: By configuring cookies as `SameSite=Strict` or `SameSite=Lax`, cross-site requests that could potentially lead to CSRF attacks are restricted.



1.5 SQLMap Analysis

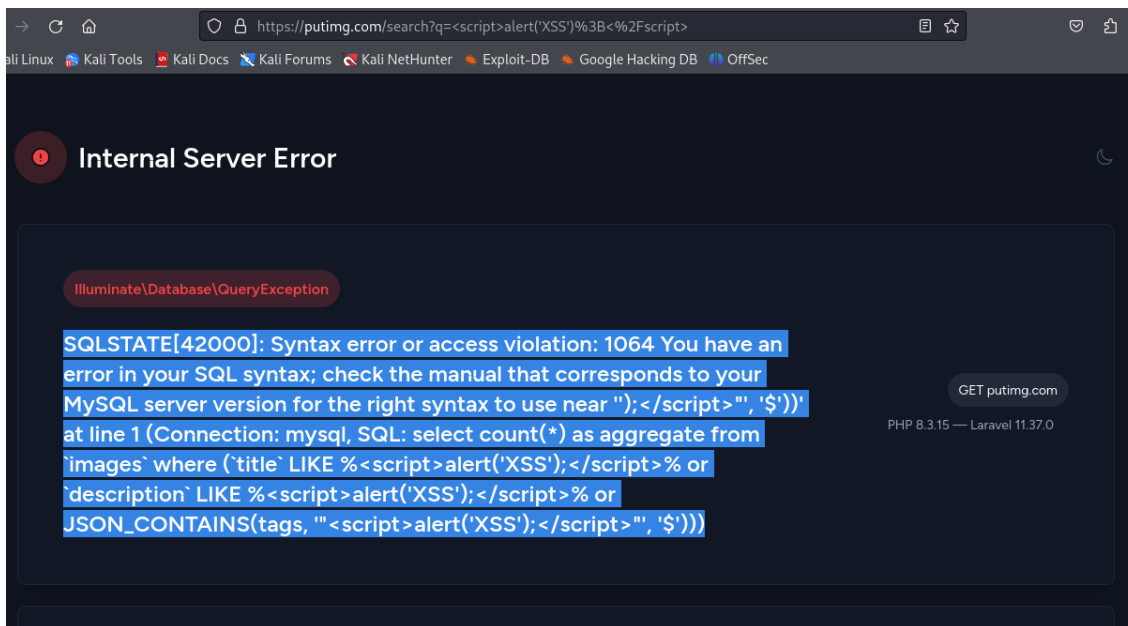
SQLMap was employed to detect SQL injection vulnerabilities in various endpoints. The following command was executed:

```
sqlmap -u "https://putimg.com/login" --data "form-email=usuario@dominio.com&form-password=contraseña" --cookie="name=value" --risk=3 --level=5 --batch --technique=BEUSTQ
```



```
kali@kali: ~  
File Actions Edit View Help  
[18:55:57] [INFO] testing 'Microsoft SQL Server/Sybase stacked queries (comment)'  
[18:55:58] [INFO] testing 'Oracle stacked queries (DBMS_PIPE.RECEIVE_MESSAGE - comment)'  
[18:55:58] [INFO] testing 'MySQL ≥ 5.0.12 AND time-based blind (query SLEEP)'  
[18:55:58] [INFO] testing 'PostgreSQL > 8.1 AND time-based blind'  
[18:55:58] [INFO] testing 'Microsoft SQL Server/Sybase time-based blind (IF)'  
[18:55:58] [INFO] testing 'Oracle AND time-based blind'  
[18:55:58] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'  
[18:55:59] [WARNING] POST parameter 'form-password' does not seem to be injectable  
[18:55:59] [CRITICAL] all tested parameters do not appear to be injectable. Try to increase values for '--level'/'--risk' options if you wish to perform more tests. Please retry with the switch '--text-only' (along with --technique=BU) as this case looks like a perfect candidate (low textual content along with inability of comparison engine to detect at least one dynamic parameter). If you suspect that there is some kind of protection mechanism involved (e.g. WAF) maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment') and/or switch '--random-agent'  
[18:55:59] [WARNING] HTTP error codes detected during run: 403 (Forbidden) - 170 times  
[18:55:59] [WARNING] your sqlmap version is outdated  
[*] ending @ 18:55:59 /2025-01-23/  
--risk=3: command not found  
(kali@kali)-[~]  
$
```

While no SQL injection vulnerabilities were identified in certain endpoints, an XSS vulnerability was detected when the payload `<script>alert('XSS');</script>` was injected into the search parameter. This indicates improper input sanitization and output encoding; might will be for the parameter LIKE.



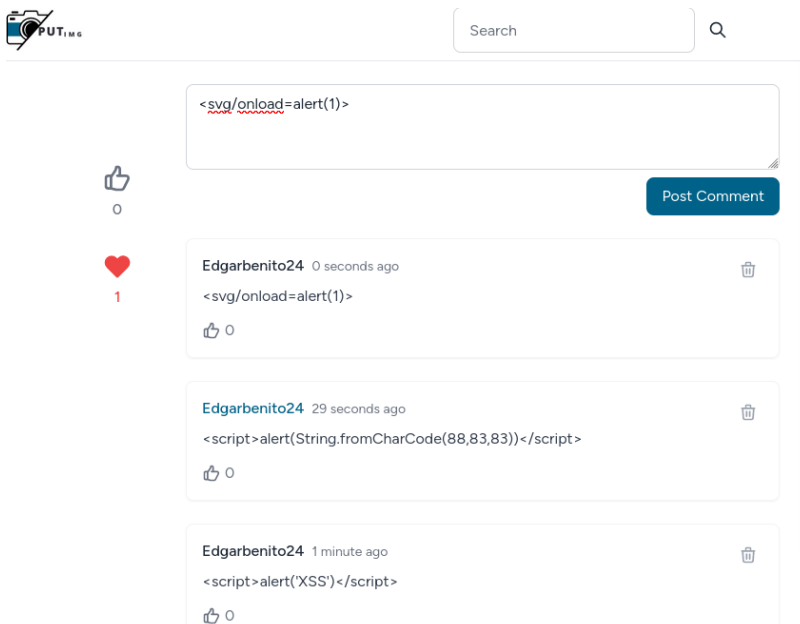
To mitigate SQL injection and XSS risks, the application should:

Use prepared statements (parameterized queries) for database interactions.

Validate and sanitize all user inputs (Example :not allowing or deleting <, >), ensuring only expected formats are accepted.

Implement output encoding to neutralize malicious scripts in dynamic content.

On the comments was not possible, that affirms more the theory about LIKE.



1.6 Dirb Directory Enumeration

Dirb was used to enumerate directories and endpoints on the server. The command executed was:

```
dirb https://putimg.com/ -o dirb results.txt
```

The scan identified several directories and files of interest:

```
/build/, /css/, /images/, /js/, /storage/
```

Pages such as /admin, /login, /register, /search, /settings, and /forgot-password.

Some of these pages' redirect or require authentication, indicating their sensitivity. To secure these directories, access controls should be reinforced, and unnecessary or unused endpoints should be removed or disabled. We finally stop dirb because can stay days for analyse all.

1.7 Curl Request Analysis

A `curl` command was used to test the response and potential issues with specific endpoints. The command executed was:

```
curl -k https://putimg.com/Qw5okq9N.notes
```

l, podría usarse para ataques dirigidos.

Solución: Verifica su contenido y, si no es necesario, elimina el arc
hivo o restringe el acceso mediante autenticación

```
[kali@kali] ~/[wapi] generated_report
└─$ curl -k https://putting.com/Qw5ok9N.notes

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>Not Found</title>

  <style>
    @! normalize.css v8.0.1 | MIT License | github.com/necolas/normalize.css v1 | line-height: 1.15; webkit-text-size-adjust: 100%; body { margin: 0; } a { background-color: transparent; } code { font-family: monospace, monospace; font-size: 1em; } [hidden] { display: none; } html { font-family: system-ui, -apple-system, BlinkMacSystemFont, Segoe UI, Roboto, Helvetica Neue, Arial, Noto Sans, sans-serif, Apple Color Emoji, Segoe UI Emoji, Segoe UI Symbol, Noto Color Emoji; line-height: 1.5; } a:after, :before { border-sizing: border-box; border: 1px solid #e2e8f0; } a { color: inherit; text-decoration: inherit; } font-family: Menlo, Monaco, Consolas, Liberation Mono, Courier New, monospace; } svg, video { display: block; vertical-align: middle; } video { max-width: 100%; height: auto; } bg-white { -bg-opacity: 1; } background-color: #fff; background-color: #255, 255, 255, var( -bg-opacity ); } bg-gray-100 { -bg-opacity: 1; } background-color: #f7fafc; background-color: rgba(247, 250, 252, var( -bg-opacity )); } border-order-200 { -border-opacity: 1; } border-color: #edf2f7; border-color: rgba(237, 247, 242, 247, var( -border-opacity )); } border-gray-400 { -border-opacity: 1; } border-color: #cbd5e0; border-color: rgba(203, 213, 224, var( -border-opacity )); } border-t { border-top: 1px; } border-r { border-right: 1px; } flex { display: flex; } grid { display: grid; } items-center { align-items: center; } justify-content { justify-content: center; } font-size { font-size: 1.25em; } h-5 { height: 1.25em; } h-8 { height: 2rem; } h-16 { height: 4rem; } text-sm { font-size: .875rem; } text-lg { font-size: 1.25rem; } leading-7 { line-height: 1.75rem; } mx-auto { margin-left: auto; margin-right: auto; } ml-1 { margin-left: .25rem; } mt-2 { margin-top: .5rem; } mr-2 { margin-right: .5rem; } ml-2 { margin-left: 1rem; } mt-4 { margin-top: 1rem; } ml-4 { margin-left: 1rem; } mt-8 { margin-top: 2rem; } ml-2 { margin-left: 1.5rem; } mt-px { margin-top: 1px; } max-w-xl { max-width: 36rem; } max-w-6xl { max-width: 72rem; } min-h-screen { min-height: 100vh; } .overflow-hidden { overflow: hidden; } p-6 { padding: 1.5rem; } py-4 { padding-top: 1rem; padding-bottom: 1rem; } px-6 { padding-left: 1.5rem; padding-right: 1.5rem; } pt-8 { padding-top: 2rem; } fixed { position: fixed; } .relative { position: relative; } top-0 { top: 0px; } right-0 { right: 0px; } .shadow { box-shadow: 0 1px 3px 0 rgba(0, 0, 0, 1), 0 1px 2px 0 rgba(0, 0, 0, .06); } text-center { text-align: center; } text-gray-200 { -text-opacity: 1; } color: #edf2f7; color: rgba(237, 247, 242, 7, var( -text-opacity )); } text-gray-300 { -text-opacity: 1; } color: #e2e8f0; color: rgba(226, 232, 240, var( -text-opacity )); } text-gray-400 { -text-opacity: 1; } color: #cbd5e0; color: rgba(203, 213, 224, var( -text-opacity )); } text-gray-500 { -text-opacity: 1; } color: #a6a6a6; color: rgba(166, 166, 166, var( -text-opacity )); }
```

The response returned a 404 error, which could have several implications:

Nonexistent File: The file may have been deleted from the server or never existed.

Access Restrictions: The file may exist but be protected by server-side access controls or firewalls, blocking direct requests.

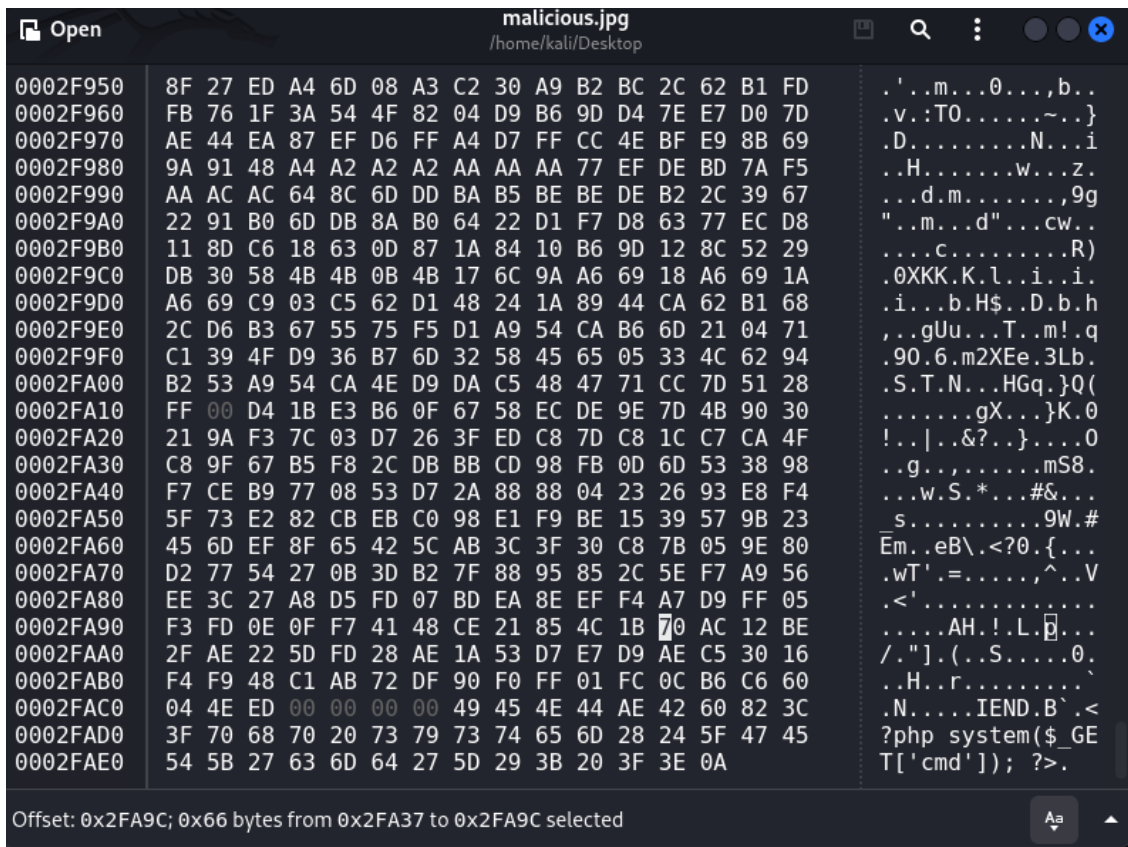
False Positives: The endpoint may have been misidentified during scanning as accessible when it is not.

Recommendation: Verify the intended resource's existence and ensure proper error handling for such requests. Access restrictions should be clearly defined to prevent unauthorized access attempts.

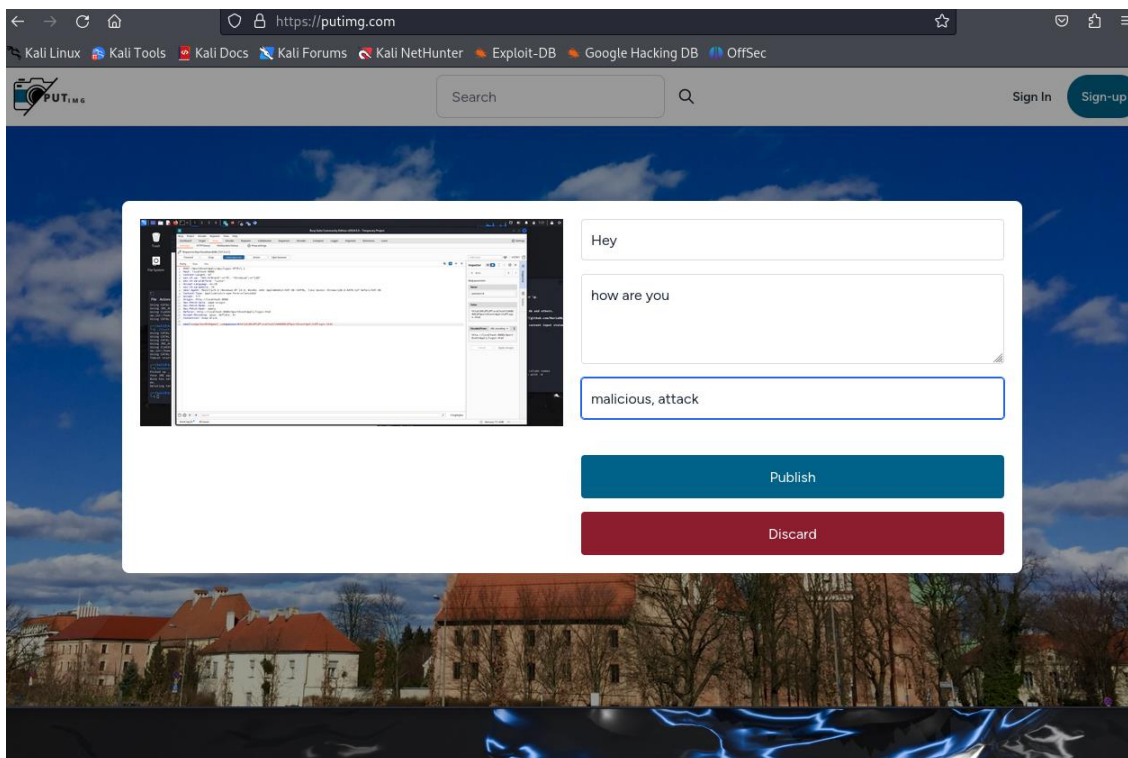
```
0{--text-opacity:1;color:#cbd5e0;color:rgba(203,213,224,var(--text-opacity))}
}
</style>
<style>
  body {
    font-family: ui-sans-serif, system-ui, -apple-system, BlinkMac
cSystemFont, "Segoe UI", Roboto, "Helvetica Neue", Arial, "Noto Sans", sans-s
erif, "Apple Color Emoji", "Segoe UI Emoji", "Segoe UI Symbol", "Noto Color E
moji";
  }
</style>
</head>
<body class="antialiased">
  <div class="relative flex items-top justify-center min-h-screen bg-gr
ay-100 dark:bg-gray-900 sm:items-center sm:pt-0">
    <div class="max-w-xl mx-auto sm:px-6 lg:px-8">
      <div class="flex items-center pt-8 sm:justify-start sm:pt-0">
        <div class="px-4 text-lg text-gray-500 border-r border-gr
ay-400 tracking-wider">
          404
        </div>
        <div class="ml-4 text-lg text-gray-500 uppercase tracking
-wider">
          Not Found
        </div>
      </div>
    </div>
  </body>
</html>
```

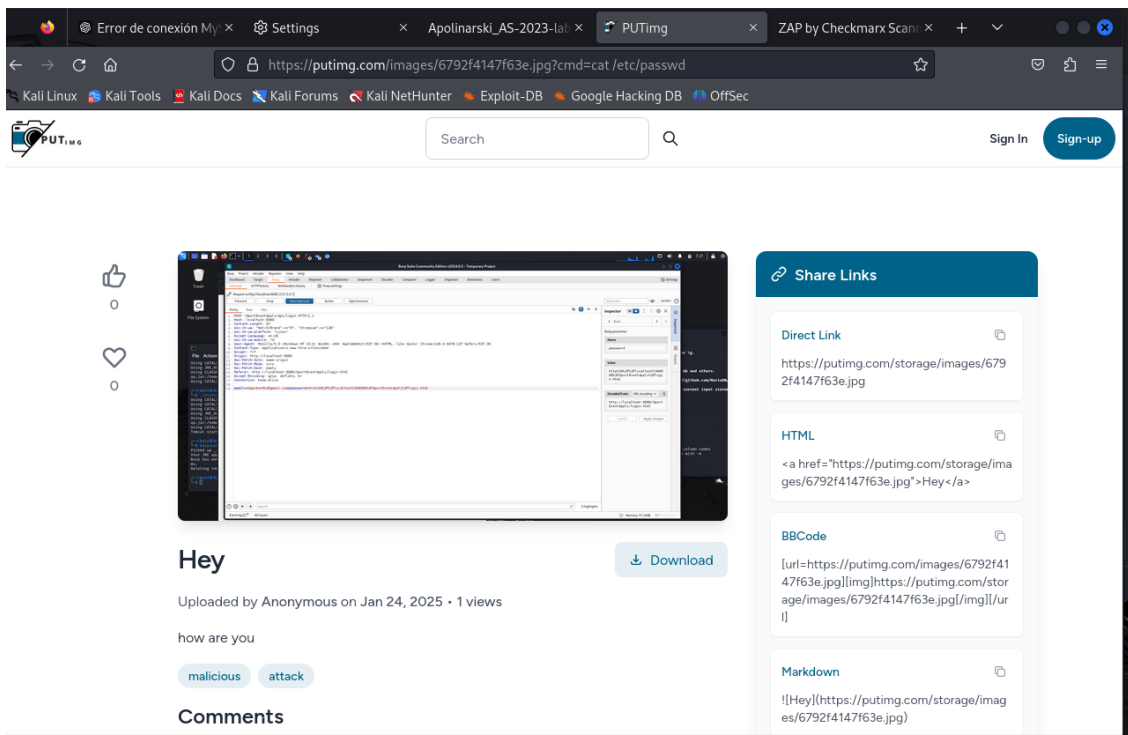
1.8 Attempt to Upload Malicious Files

Attempts were made to upload potentially malicious files to the application to assess its file upload handling mechanisms. Initially, an image file was edited at the hex level, embedding the script `<?php system($_GET['cmd']); ?>`. The modified file was then uploaded to the application.



Surprisingly, the application accepted the file without rejection and treated it as a valid JPG image. However, upon attempting to execute the embedded PHP script, the script failed to function, and no malicious actions could be performed.





This outcome indicates that while the application does not reject suspicious content, it effectively disallows the execution of PHP scripts. To strengthen this defense, the application should implement robust content inspection mechanisms to validate file integrity. Additional measures include ensuring that executable code is entirely blocked, employing MIME type validation, and scanning for potentially malicious content using antivirus tools during the upload process.

1. 9 Recommendations and Conclusion

The application demonstrates several critical and medium-risk vulnerabilities that require immediate attention. The following measures should be prioritized:

1. **Security Headers:** Configure HSTS, CSP, and other security headers to mitigate XSS, MITM, and MIME sniffing attacks.
2. **Input Validation:** Enforce strict validation for all user inputs to prevent SQL injection, XSS, and other injection-based attacks.
3. **Secure Cookies:** Mark all session and CSRF-related cookies with the `HttpOnly` and `Secure` flags.
4. **Periodic Scanning:** Regularly conduct security scans using tools like OWASP ZAP, Nikto, and SQLMap to identify and fix vulnerabilities proactively.

Addressing these issues will significantly enhance the security posture of the application and protect user data from potential attacks. Further testing and audits are recommended after implementing these changes to ensure all vulnerabilities are adequately addressed.

2. White Box

2.1 Laravel Controllers

2.1.1 ImageController

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use App\Models\Image;
6
7 class ImageController extends Controller
8 {
9     public function show($filename)
10     {
11         $image = Image::where('filename', $filename)->firstOrFail();
12         $uploader = $image->user_id === 0 ? 'Anonymous' : User::find($image->user_id)->name
13
14         return view('livewire.show-post', compact('image', 'uploader'));
15     }
16 }
17
```

Vulnerability: Access to User Data Without Validation

Affected Line: 12

Code:

```
$uploader = $image->user_id === 0 ? 'Anonymous' : User::find($image->user_id)->name;
```

Issue: If `User::find` returns null (e.g., when the `user_id` is invalid or the user does not exist), attempting to access the `name` property will cause an error.

Mitigation: Add a null check for the user object before accessing its properties:

```
$uploader = $image->user_id === 0 ? 'Anonymous' : optional(User::find($image->user_id))->name ?? 'Unknown';
```

2.1.2 ProfileController

```
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use App\Http\Requests\ProfileUpdateRequest;
6  use Illuminate\Http\RedirectResponse;
7  use Illuminate\Http\Request;
8  use Illuminate\Support\Facades\Auth;
9  use Illuminate\Support\Facades\Redirect;
10 use Illuminate\View\View;
11
12 class ProfileController extends Controller
13 {
14     /**
15      * Display the user's profile form.
16      */
17     public function edit(Request $request): View
18     {
19         return view('profile.edit', [
20             'user' => $request->user(),
21         ]);
22     }
23
24     /**
25      * Update the user's profile information.
26      */
27     public function update(ProfileUpdateRequest $request): RedirectResponse
28     {
29         $request->user()->fill($request->validated());
```



```

31         if ($request->user()->isDirty('email')) {
32             $request->user()->email_verified_at = null;
33         }
34
35         $request->user()->save();
36
37         return Redirect::route('profile.edit')->with('status', 'profile-updated');
38     }
39
40     /**
41      * Delete the user's account.
42      */
43     public function destroy(Request $request): RedirectResponse
44     {
45         $request->validateWithBag('userDeletion', [
46             'password' => ['required', 'current_password'],
47         ]);
48
49         $user = $request->user();
50
51         Auth::logout();
52
53         $user->delete();
54
55         $request->session()->invalidate();
56         $request->session()->regenerateToken();
57
58         return Redirect::to('/');
59     }
60 }

```

Affected Line: 22

Code:

```
$request->user()->fill($request->validated());
```

Issue: If the ProfileUpdateRequest validation rules are not strict, users could manipulate data to update unauthorized fields.

Mitigation: Ensure the ProfileUpdateRequest class explicitly defines the allowed fields:

Vulnerability 2: Dependency on Secure Password Hashing

Affected Lines: 33-37

Code:

```

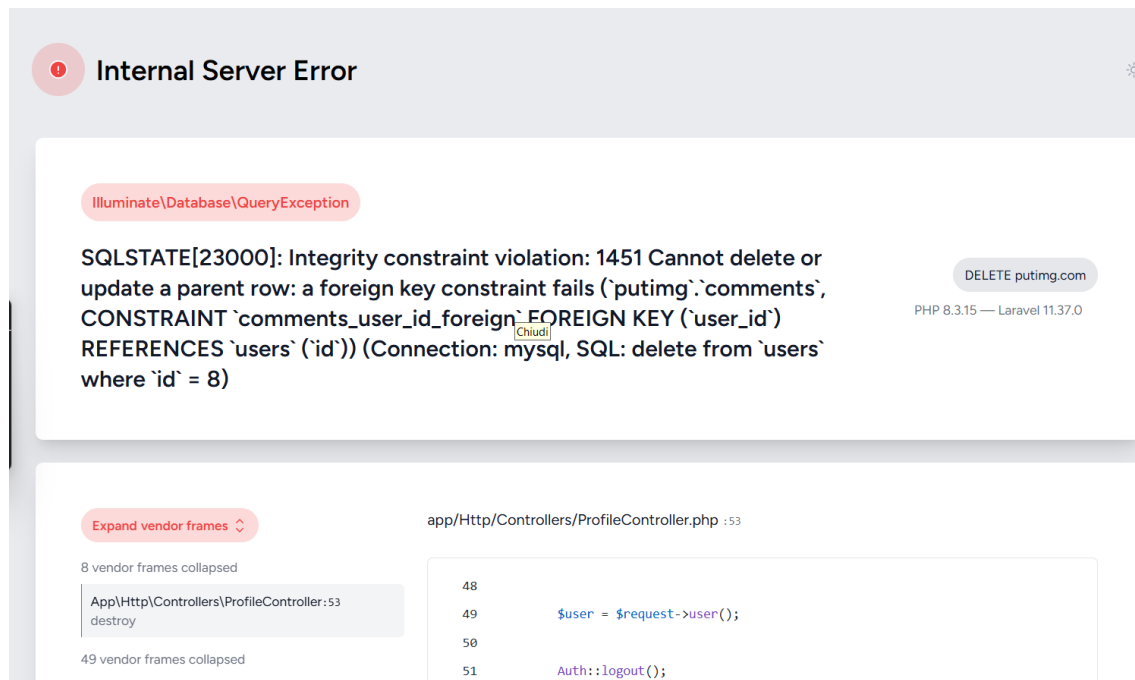
$request->validateWithBag('userDeletion', [
    'password' => ['required', 'current_password'],
]);

```

Issue: This relies on the password hashing configuration being secure. Weak or outdated hashing algorithms could compromise security.

Mitigation: Ensure the application uses modern hashing algorithms such as bcrypt or Argon2. Laravel's default setup already uses bcrypt, but configurations should be reviewed periodically.

Database Problem:



The error message stems from a foreign key constraint violation in the database. Specifically, the comments table is linked to the users table via the user_id foreign key, which prevents deletion of a user who still has associated records in the comments table.

It seems that there is an assumption that a primary key would be in place, but after reviewing the code and database structure, it appears that the necessary primary key wasn't set. This oversight is likely causing the inability to delete a user account properly. As a result, when a user is deleted, they may still be able to access their account and upload photos, even though their account should be deleted.

In summary:

The primary key on the users table seems to be missing or misconfigured.

The foreign key relationship between comments and users prevents deletion if there are dependent records.

This leads to a situation where users can still upload photos after their account is deleted.

2.1.3 UploadController

```
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request;
6  use Illuminate\Support\Facades\Storage;
7  use App\Models\Image;
8
9  class UploadController extends Controller
10 {
11     public function store(Request $request)
12     {
13         try {
14             if (!$request->hasFile('image')) {
15                 return response()->json(['error' => 'No file uploaded'], 400);
16             }
17
18             $file = $request->file('image');
19
20             // Check MIME type
21             $mimeType = $file->getMimeType();
22             $allowedTypes = ['image/jpeg', 'image/png', 'image/gif', 'image/webp'];
23
24             if (!in_array($mimeType, $allowedTypes)) {
25                 return response()->json([
26                     'error' => 'Only image files are allowed (PNG, JPEG, GIF, WEBP)'
27                 ], 400);
28             }
29         }
30     }
31 }
```

```

29
30 // Size validation (32MB)
31 if ($file->getSize() > 32 * 1024 * 1024) {
32     return response()->json([
33         'error' => 'File size must be less than 32MB'
34     ], 400);
35 }
36
37 $filename = $file->getClientOriginalName();
38 $folder = uniqid() . '_' . time();
39
40 try {
41     Storage::disk('public')->makeDirectory('images/tmp/' . $folder);
42     $file->storeAs('images/tmp/' . $folder, $filename, 'public');
43
44     if (!Storage::disk('public')->exists("images/tmp/{$folder}/{$filename}")) {
45         throw new \Exception('Failed to save file');
46     }
47
48     return response()->json([
49         'folder' => $folder,
50         'filename' => $filename
51     ]);
52 } catch (\Exception $e) {
53     \Log::error('Storage error: ' . $e->getMessage());
54     return response()->json([
55         'error' => 'Failed to save file'

```

```

56         'error' => 'Failed to save file'
57     ], 500);
58 }
59
60 } catch (\Exception $e) {
61     \Log::error('Upload error: ' . $e->getMessage());
62     return response()->json([
63         'error' => 'Upload failed'
64     ], 500);
65 }
66 }
67
68 public function delete(Request $request)
69 {
70     $folder = $request->input('folder');
71     if ($folder) {
72         Storage::disk('public')->deleteDirectory('/images/tmp/' . $folder);
73         return response()->json(['message' => 'Folder deleted']);
74     }
75     return response()->json(['error' => 'No folder specified'], 400);
76 }
77
78
79 public function publish(Request $request)
80 {
81     try {
82         $validated = $request->validate([
83             'title' => 'required|string',

```

```

82     $validated = $request->validate([
83         'title' => 'required|string',
84         'description' => 'nullable|string',
85         'tags' => 'nullable|array',
86         'imagePath' => 'required|string'
87     ]);
88
89     // Decode the URL-encoded path
90     $path = urldecode(parse_url($request->imagePath, PHP_URL_PATH));
91     $pathParts = explode('/', $path);
92
93     $storageIndex = array_search('storage', $pathParts);
94     $folder = $pathParts[$storageIndex + 3];
95     $filename = $pathParts[$storageIndex + 4];
96
97     // Decode the filename as well
98     $filename = urldecode($filename);
99
100    \Log::info("Looking for file in: images/tmp/{$folder}/{filename}");
101
102    // Verify file exists before proceeding
103    if (!Storage::disk('public')->exists("images/tmp/{$folder}/{filename}")) {
104        \Log::error("File not found in: images/tmp/{$folder}/{filename}");
105        \Log::error("Available files in folder: " . json_encode(Storage::disk('public')->files("images/tmp
106        return response()->json([
107            'success' => false,
108            'message' => 'Source file not found'
109        ], 404);
110    }

```

```

112    $newFilename = uniqid() . '.' . pathinfo($filename, PATHINFO_EXTENSION);
113
114    // Ensure directories exist
115    Storage::disk('public')->makeDirectory('images');
116
117    // Move file
118    $moved = Storage::disk('public')->move(
119        "images/tmp/{$folder}/{filename}",
120        "images/{$newFilename}"
121    );
122
123    // Remove temporal folder
124    Storage::disk('public')->deleteDirectory('images/tmp/' . $folder);
125
126    if (!$moved) {
127        return response()->json([
128            'success' => false,
129            'message' => 'Failed to move file'
130        ], 500);
131    }
132
133    $image = Image::create([
134        'title' => $validated['title'],
135        'description' => $validated['description'],
136        'user_id' => auth()->id() ?? 0,
137        'date' => now(),
138        'format' => pathinfo($filename, PATHINFO_EXTENSION),
139        'tags' => $validated['tags'] ?? [],
140        'filename' => $newFilename

```

```

142
143         return response()->json([
144             'success' => true,
145             'redirect' => route('image.show', $newFilename)
146         ]);
147     } catch (\Exception $e) {
148         \Log::error('Error in publish: ' . $e->getMessage());
149         return response()->json([
150             'success' => false,
151             'message' => $e->getMessage()
152         ], 500);
153     }
154 }
155
156 public function destroy($filename)
157 {
158     $image = Image::where('filename', $filename)->firstOrFail();
159
160     // Check if user owns the image
161     if ($image->user_id !== auth()->id()) {
162         abort(403);
163     }
164
165     // Delete file from storage
166     Storage::delete('public/images/' . $filename);
167
168     // Delete from database
169     $image->delete();

```

```

169     $image->delete();
170
171     return redirect()->route('home')->with('success', 'Image deleted successfully');
172 }
173
174 }
175

```

Vulnerability 1: Incomplete MIME Type Validation

Affected Line: 19

Code:

```
$mimeType = $file->getMimeType();
```

```
$allowedTypes = ['image/jpeg', 'image/png', 'image/gif', 'image/webp'];
```

Issue: Attackers can upload malicious files with spoofed MIME types, bypassing this check.

Vulnerability 2: Generic Exception Handling

Affected Line: 48

Code:

```
throw new \Exception('Failed to save file');
```

Issue: Generic error messages make debugging difficult and provide no actionable information.

Mitigation: Use specific exceptions and log detailed error messages:

```
throw new FileStorageException('Failed to save file: ' . $e->getMessage());
```

Vulnerability 3: Directory Traversal Attack

Affected Line: 99

Code:

```
$path = urldecode(parse_url($request->imagePath, PHP_URL_PATH));
```

Issue: An attacker could exploit this to access files outside the intended directory using traversal sequences (e.g., ../../).

Vulnerability 4: Missing Authorization for File Deletion

Affected Line: 200

Code:

```
Storage::delete('public/images/' . $filename);
```

Issue: No check ensures the user attempting to delete the file is its owner.

Mitigation: Add an ownership check before deleting the file:

File	Line	Issue	Mitigation
ImageController	12	Accessing a user without verifying its existence.	Ensure that User::find returns a valid object before accessing name.
ProfileController	22	Potentially insufficient input validation.	Ensure that validation rules are complete and rigorous.
ProfileController	33	Dependency on properly configured password hashing.	Ensure the use of modern hashing algorithms (bcrypt, Argon2).
UploadController	19	Insufficient MIME type checking.	Also verify the file extension and content.
UploadController	48	Generic exception handling.	Provide more detailed error messages without exposing sensitive information.
UploadController	99	Potential directory traversal through image path.	Normalize and validate the path to prevent unauthorized access.
UploadController	200	Lack of authorization for file deletion.	Verify that the user is authorized before deleting the file.

2.2 Middleware

2.2.1 EnsureEmailIsVerified.php

```
1 <?php
2
3 namespace App\Http\Middleware;
4
5 class EnsureEmailIsVerified
6 {
7     public function handle($request, Closure $next)
8     {
9         if (!$request->user() ||
10             ($request->user() instanceof MustVerifyEmail &&
11              !$request->user()->hasVerifiedEmail())) {
12             return $request->expectsJson()
13                 ? abort(403, 'Your email address is not verified.')
14                 : Redirect::guest(URL::route('verification.notice'));
15         }
16
17         return $next($request);
18     }
19 }
```

Potential Vulnerabilities:

1. Line 8-11:
 - **Vulnerability:** The middleware checks whether the user implements the MustVerifyEmail interface. However, there is no explicit safeguard against non-authenticated users or edge cases where `$request->user()` is manipulated or null.
 - **Risk:** If MustVerifyEmail is accidentally omitted or the user object is tampered with, the middleware may fail silently, exposing routes to unverified users.
 - **Mitigation:** Add a strict validation to check that `$request->user()` always adheres to the intended type (e.g., authenticated User models only).
2. Line 12-13:
 - **Vulnerability:** The middleware uses `abort(403)` for JSON requests, but this does not provide any recovery mechanism or detailed information beyond the error code. This might confuse the frontend handling JSON responses.
 - **Risk:** Poor user experience or debugging difficulties if the 403 response isn't properly handled.
 - **Mitigation:** Include detailed error messages or standardized error structures for JSON responses.

Recommended Fix:

- Validate `$request->user()` strictly to ensure it is always an authenticated and valid user object.
- Improve error handling for JSON responses to include helpful details.

2.2.2 RedirectIfNotVerified.php

```
1 <?php
2
3 namespace App\Http\Middleware;
4
5 use Closure;
6 use Illuminate\Http\Request;
7 use Illuminate\Support\Facades\Auth;
8
9 class RedirectIfNotVerified
10 {
11     public function handle(Request $request, Closure $next)
12     {
13         if (!Auth::user()->hasVerifiedEmail()) {
14             Auth::logout();
15             return redirect()->route('login')
16                 ->with('error', 'You must verify your email address before logging in. Please check your email for
17         }
18
19         return $next($request);
20     }
21 }
```

Potential Vulnerabilities:

Line 9:

Vulnerability: Directly accessing `Auth::user()->hasVerifiedEmail()` without validating if `Auth::user()` exists can cause a **null pointer exception** if the user is not logged in or the `Auth::user()` is unexpectedly null.

Risk: This could lead to runtime errors if the middleware is executed for unauthenticated requests.

Mitigation: Add a null check for `Auth::user()` before calling `hasVerifiedEmail()`.

Line 10-13:

Vulnerability: Logging out users and redirecting them without explicitly recording their session details or providing an audit trail may lead to difficulties in tracking abuse or debugging.

Risk: Potential exploitation of sessions and lack of transparency in why the user was logged out.

Mitigation: Log an event or store audit details (e.g., IP address, timestamp, action) when logging a user out due to an unverified email.

1. Middleware Line Vulnerability Description Mitigation			
EnsureEmailsVerified	8-11	Lack of strict validation for <code>\$request->user()</code> and interface checking	Add type checks and strict validation for user objects.
EnsureEmailsVerified	12-13	Incomplete error handling for JSON responses	Provide standardized error structures or more informative responses.
RedirectIfNotVerified	9	Potential null pointer exception for <code>Auth::user()</code>	Add null checks for <code>Auth::user()</code> before accessing methods.
RedirectIfNotVerified	10-13	Lack of logging for user session invalidation	Log session invalidation or add an audit trail for logout actions.

2.3 Livewire Files

2.3.1 LoginForm.php

```

1  <?php
2
3  namespace App\Livewire\Forms;
4
5  use Illuminate\Auth\Events\Lockout;
6  use Illuminate\Support\Facades\Auth;
7  use Illuminate\Support\Facades\RateLimiter;
8  use Illuminate\Support\Str;
9  use Illuminate\Validation\ValidationException;
10 use Livewire\Attributes\Validate;
11 use Livewire\Form;
12
13 class LoginForm extends Form
14 {
15     #[Validate('required|string|email')]
16     public string $email = '';
17
18     #[Validate('required|string')]
19     public string $password = '';
20
21     #[Validate('boolean')]
22     public bool $remember = false;
23
24     /**
25      * Attempt to authenticate the request's credentials.
26      *
27      * @throws \Illuminate\Validation\ValidationException
28      */
29     public function authenticate(): void
30     {

```

```

31     $this->ensureIsNotRateLimited();
32
33     if (! Auth::attempt($this->only(['email', 'password']), $this->remember)) {
34         RateLimiter::hit($this->throttleKey());
35
36         throw ValidationException::withMessages([
37             'form.email' => trans('auth.failed'),
38         ]);
39     }
40
41     RateLimiter::clear($this->throttleKey());
42 }
43
44 /**
45  * Ensure the authentication request is not rate limited.
46  */
47 protected function ensureIsNotRateLimited(): void
48 {
49     if (! RateLimiter::tooManyAttempts($this->throttleKey(), 5)) {
50         return;
51     }
52
53     event(new Lockout(request()));
54
55     $seconds = RateLimiter::availableIn($this->throttleKey());
56
57     throw ValidationException::withMessages([
58         'form.email' => trans('auth.throttle', [
59             'seconds' => $seconds,
60             'minutes' => ceil($seconds / 60),

```

```

61         ]),
62     ]);
63 }
64
65 /**
66  * Get the authentication rate limiting throttle key.
67  */
68 protected function throttleKey(): string
69 {
70     return Str::transliterate(Str::lower($this->email).'|'.request()->ip());
71 }
72 }
73

```

Line 23: The authenticate method processes user inputs directly (email and password). If these inputs are not properly sanitized, they could be exploited for injection attacks, such as SQL injection or command injection.

Mitigation: Use Laravel's validation rules or sanitize inputs before processing them.

Line 29: The RateLimiter mechanism is used to prevent brute force attacks; however, it relies on a predefined limit (5 attempts) and does not incorporate advanced protections like CAPTCHA.

Mitigation: Add CAPTCHA verification after repeated login failures to prevent automated brute force attacks.

Line 48: The `throttleKey` method generates a rate limiter key based on the email and request IP address. This approach may expose predictable keys, allowing attackers to bypass rate limits.

Mitigation: Use a hashed or encrypted key for better security, combining unique identifiers such as user agents.

2.3.2 Logout.php

```
1  <?php
2
3  namespace App\Livewire\Actions;
4
5  use Illuminate\Support\Facades\Auth;
6  use Illuminate\Support\Facades\Session;
7
8  class Logout
9  {
10     /**
11      * Log the current user out of the application.
12      */
13     public function __invoke(): void
14     {
15         Auth::guard('web')->logout();
16
17         Session::invalidate();
18         Session::regenerateToken();
19     }
20 }
```

Line 13: The `Session::invalidate()` and `Session::regenerateToken()` methods are called to terminate the session. However, this assumes that all tokens are securely invalidated, which might not cover distributed sessions or sessions on multiple devices.

Mitigation: Ensure all active user sessions across devices are terminated during logout.

2.3.3 Summary Table of Issues and Mitigation Strategies

File	Line	Issue	Mitigation
LoginForm.php	23	Possibility of injection attacks during login.	Ensure inputs (<code>email</code> and <code>password</code>) are sanitized before processing.
LoginForm.php	29	Potential brute force login attempts.	Implement CAPTCHA after multiple failed attempts and limit login attempts further.
LoginForm.php	48	Rate limiter key vulnerable to predictable attacks.	Use a secure and unpredictable throttle key, such as a hashed combination of email and IP.
Logout.php	13	Session invalidation requires verification.	Confirm that all active user sessions are terminated and tokens are securely regenerated.

2.4 Models

2.4.1 Comment.php

```
1  <?php
2  namespace App\Models;
3
4  use Illuminate\Database\Eloquent\Model;
5
6  class Comment extends Model
7  {
8      protected $primaryKey = 'comment_id';
9      public $timestamps = false;
10
11     protected $fillable = ['user_id', 'image_id', 'content', 'date', 'like_count'];
12
13     protected $casts = [
14         'date' => 'datetime'
15     ];
16
17     public function user()
18     {
19         return $this->belongsTo(User::class);
20     }
21
22     public function image()
23     {
24         return $this->belongsTo(Image::class, 'image_id', 'image_id');
25     }
26 }
```

Line 9: \$fillable is defined but does not specify any safeguards against malicious input for fields like content. This can lead to SQL injection if the content field is not validated or sanitized before being stored.

Mitigation: Implement validation at the controller or service layer to sanitize and validate all user inputs for content.

Line 19: The image relationship uses the image_id field. If there are unvalidated inputs for this ID in related operations, it may lead to unintended access or SQL injection attacks.

Mitigation: Validate image_id inputs rigorously and ensure that they are integers or valid keys before querying the database.

2.4.2 CommentLike.php

```
1 <?php
2
3 namespace App\Models;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class CommentLike extends Model
8 {
9     public $timestamps = false;
10    protected $fillable = ['user_id', 'comment_id'];
11
12    public function comment()
13    {
14        return $this->belongsTo(Comment::class, 'comment_id', 'comment_id');
15    }
16
17    public function user()
18    {
19        return $this->belongsTo(User::class);
20    }
21 }
```

Line 8: \$fillable allows direct assignment of user_id and comment_id. Without proper validation, these fields could be abused, such as attempting to forge likes or manipulate data.

Mitigation: Enforce validation on user_id and comment_id in the controller or service layer, ensuring the data is legitimate and not tampered with.

Line 12: The comment relationship queries based on comment_id. If this value is derived from user inputs without validation, it could lead to query manipulation or unintended results.

Mitigation: Sanitize and validate comment_id inputs to ensure they match existing comments.

2.4.3 Image.php

```
1  <?php
2
3  namespace App\Models;
4
5  use Illuminate\Database\Eloquent\Model;
6
7  class Image extends Model
8  {
9      protected $primaryKey = 'image_id';
10     public $timestamps = false;
11
12     protected $fillable = [
13         'title',
14         'description',
15         'user_id',
16         'date',
17         'format',
18         'views',
19         'tags',
20         'filename'
21     ];
22
23     protected $casts = [
24         'tags' => 'array',
25         'date' => 'datetime'
26     ];
27
28     public function user()
29     {
30         return $this->belongsTo(User::class);
31     }
32
33     public function likes()
34     {
35         return $this->hasMany(ImageLike::class, 'image_id', 'image_id');
36     }
37
38     public function favorites()
39     {
40         return $this->hasMany(ImageFavorite::class, 'image_id', 'image_id');
41     }
42
43     public function comments()
44     {
45         return $this->hasMany(Comment::class, 'image_id', 'image_id');
46     }
47
48     public function favoritedBy()
49     {
50         return $this->belongsToMany(User::class, 'image_favorites', 'image_id', 'user_id', 'image_id');
51     }
52
53     public function likedBy()
54     {
55         return $this->belongsToMany(User::class, 'image_likes', 'image_id', 'user_id', 'image_id');
56     }
57
58 }
```

Line 12: \$fillable includes sensitive fields like filename. Direct assignment without validation could lead to path traversal or file system vulnerabilities.

Mitigation: Ensure filename inputs are sanitized and validated for safe file paths and prevent directory traversal attacks.

Line 14: The tags field is cast as an array. If this field is derived from user input, malicious tags could be injected into the database and later exploited.

Mitigation: Validate tags as an array with specific constraints, such as disallowing certain characters or formats.

Line 27: The comments relationship uses image_id to query comments. If image_id is not properly validated when passed as input, it can lead to SQL injection or unintended results.

Mitigation: Validate image_id inputs rigorously, ensuring they are integers or valid database keys.

File Line Issue Mitigation

Comment.php 9	Fields like content may not be sanitized, leading to SQL injection.	Validate and sanitize content at the controller or service layer.
Comment.php 19	Unvalidated image_id in the image relationship.	Ensure image_id is an integer and validate inputs before querying the database.
CommentLike.php 8	fillable fields like user_id and comment_id are not validated.	Enforce validation on user_id and comment_id in the controller or service layer.
CommentLike.php 12	Unvalidated comment_id in the comment relationship.	Sanitize and validate comment_id inputs to match existing comments.
Image.php 12	Direct assignment to filename without validation.	Sanitize and validate filename to prevent directory traversal and unsafe file paths.
Image.php 14	tags field may allow malicious data injection.	Validate tags as an array with specific constraints to prevent injection attacks.
Image.php 27	Unvalidated image_id in the comments relationship.	Validate image_id inputs to ensure they are safe and match existing records.

2.5 Requests

2.5.1 CustomEmailVerificationRequest.php

```
1  <?php
2
3  namespace App\Http\Requests;
4
5  use App\Models\User;
6  use Illuminate\Foundation\Http\FormRequest;
7  use Illuminate\Support\Facades\Hash;
8
9  class CustomEmailVerificationRequest extends FormRequest
10 {
11     public function authorize()
12     {
13         $user = \App\Models\User::find($this->route('id'));
14
15         if (! $user) {
16             return false;
17         }
18
19         // Compare the hash in the URL with the expected hash
20         $expectedHash = sha1($user->getEmailForVerification());
21
22         if (! hash_equals($this->route('hash'), $expectedHash)) {
23             return false;
24         }
25
26         return true;
27     }
28
29     public function rules()
30     {
```

Line 11-13: Potential Vulnerability in User ID Handling

Issue: The function `\App\Models\User::find($this->route('id'))` retrieves the user by the ID passed in the route. If the ID is easily predictable or exposed, an attacker could manipulate it in the URL to gain information about other users.

Vulnerability: Predictable User ID or Lack of protection against unauthorized access. The user ID should not be easily manipulable to avoid unauthorized access.

Improvement: It's better to use a more secure identifier, such as a unique token, instead of a predictable user ID, or to add extra access control to prevent unauthorized users from manipulating the route.

Line 16-18: Use of sha1() for Hash Comparison

Issue: The use of `sha1()` for hash comparison is outdated and weak. SHA-1 is vulnerable to collision attacks, and is no longer considered secure for cryptographic purposes.

Vulnerability: Weak Hashing Algorithm (SHA-1).

Improvement: It is recommended to use stronger hashing algorithms, like `sha256`, or better yet, use cryptographic functions specifically designed for secure comparisons, such as `bcrypt`.

2.5.2 ProfileUpdateRequest.php

```
1  <?php
2
3  namespace App\Http\Requests;
4
5  use App\Models\User;
6  use Illuminate\Foundation\Http\FormRequest;
7  use Illuminate\Validation\Rule;
8
9  class ProfileUpdateRequest extends FormRequest
10 {
11     /**
12      * Get the validation rules that apply to the request.
13      *
14      * @return array<string, \Illuminate\Contracts\Validation\ValidationRule|array<mixed>|string>
15      */
16     public function rules(): array
17     {
18         return [
19             'name' => ['required', 'string', 'max:255'],
20             'email' => [
21                 'required',
22                 'string',
23                 'lowercase',
24                 'email',
25                 'max:255',
26                 Rule::unique(User::class)->ignore($this->user()->id),
27             ],
28         ];
29     }
30 }
```

Line 15: Insufficient Authentication Check

Issue: There is no evident check that the user is authenticated before accessing this request. The function `user()` might return null if the user is not authenticated, leading to validation rules being processed improperly.

Vulnerability: Lack of Authentication Check. If the user is not authenticated, the validation might still proceed, potentially causing unauthorized users to modify profile information.

Improvement: Ensure that the user is authenticated before processing the request by using middleware like `auth`.

Line 16: unique Rule Without Full Check

Issue: The rule `Rule::unique(User::class)->ignore($this->user()->id)` is good for ensuring that the user's email is unique but it only ignores the user's current email. However, it assumes that no other part of the system is incorrectly handling the uniqueness check.

Vulnerability: Possible Email Overwrite Risk. If other parts of the application don't enforce uniqueness, an attacker might exploit this rule to modify an email that conflicts with another user's email.

Improvement: In addition to this rule, ensure explicit uniqueness checks in all parts of the application (including the controller action) and handle possible race conditions when updating the email.

Line 19: Missing Validation for Sensitive Data (e.g., Passwords, Roles)

Issue: There is no validation for other sensitive fields like password or user roles. If a user modifies their password or role without proper validation, this could lead to security issues.

Vulnerability: Unprotected Modification of Sensitive Data. Critical fields like passwords or roles should be validated and handled with care.

Improvement: Add specific validation for sensitive fields like passwords to ensure they are properly validated (e.g., minimum length, strong characters) and handled securely.

2.6 Kernel.php

```
1  <?php
2
3  namespace App\Http;
4
5  use App\Http\Middleware\RedirectIfNotVerified;
6  use Illuminate\Auth\Middleware\EnsureEmailIsVerified;
7  use Illuminate\Foundation\Http\Kernel as HttpKernel;
8  use Illuminate\Routing\Middleware\ValidateSignature;
9
10 class Kernel extends HttpKernel
11 {
12     protected $middlewareGroups = [
13         'web' => [
14             // ... other middleware
15         ],
16     ];
17
18     protected $routeMiddleware = [
19         // ... other middleware
20         'verified.login' => RedirectIfNotVerified::class,
21         'verified' => EnsureEmailIsVerified::class,
22         'signed' => ValidateSignature::class,
23     ];
24 }
```

Vulnerabilities and Configuration Issues:

1. Missing Global Middleware (Line 7 - 12)

Issue: The web middleware group is defined, but there is no explicit mention of **security-related middleware** such as `\Illuminate\Session\Middleware\StartSession` or `\Illuminate\View\Middleware\ShareErrorsFromSession`. These middleware are typically essential for managing session and CSRF protection.

Potential Vulnerability: Lack of session and CSRF protection.

RedirectIfNotVerified Middleware (Line 17)

Issue: This middleware (`RedirectIfNotVerified::class`) is used to redirect users who are not verified. However, the logic might expose user data or prevent users from gaining access to important resources if not properly handled. If not properly configured, this could cause issues with user access or reveal certain data unintentionally.

Potential Vulnerability: Improper handling of unverified users.

Recommendation: Review the RedirectIfNotVerified middleware to ensure that it's only allowing access to resources that are appropriately restricted. Also, confirm that it doesn't inadvertently expose user data to unauthorized parties during the redirection process.

3. EnsureEmailIsVerified Middleware (Line 18)

Issue: This middleware checks that the user's email is verified before accessing certain routes. While this is important for protecting access to sensitive areas of the app, there could be misconfigurations in routes where this middleware is applied, allowing unauthorized access or restricting users unnecessarily.

Potential Vulnerability: Incorrect application of email verification requirement. If it's applied to routes where email verification should not be a requirement, users might face unnecessary restrictions.

Recommendation: Make sure that EnsureEmailIsVerified is applied only to routes that genuinely require email verification (e.g., after registration or password reset). For any routes that do not require email verification, ensure that the middleware is not mistakenly applied.

4. ValidateSignature Middleware (Line 19)

Issue: The ValidateSignature middleware is used to ensure that the request URL matches a valid signature. This is typically used to protect actions such as password resets or email verification links from tampering. However, if this middleware is misapplied or the signature generation is weak, attackers could potentially bypass it by generating valid signatures.

Potential Vulnerability: Weak or Misconfigured URL Signatures.

Recommendation: Ensure that the ValidateSignature middleware is used only for routes that require signed URLs. Also, make sure that the signature generation and validation logic is securely implemented. Laravel's default `URL::signedRoute()` method can be used for securely generating signed URLs.

2.6.1 Summary Table

Vulnerability/Issue	Line	Description	Recommendation
Missing Session and CSRF Protection Middleware	Lines 7-12	The web middleware group is defined, but there is no mention of essential session management and CSRF protection middleware, which are crucial for security.	Add middleware like <code>StartSession</code> , <code>ShareErrorsFromSession</code> , <code>VerifyCsrfToken</code> , etc., to the web group to ensure proper protection.
Improper Handling of Unverified Users	Line 17	The <code>RedirectIfNotVerified</code> middleware might expose user data or prevent access to resources unintentionally if misconfigured.	Review the logic of the <code>RedirectIfNotVerified</code> middleware to ensure proper handling without exposing sensitive information.
Incorrect Application of Email Verification	Line 18	The <code>EnsureEmailIsVerified</code> middleware may be applied to routes where email verification is not necessary, potentially blocking legitimate access.	Apply the <code>EnsureEmailIsVerified</code> middleware only to routes that truly require email verification.
Weak or Misconfigured URL Signature Handling	Line 19	The <code>ValidateSignature</code> middleware is used for signed URLs, but weak signature generation or misapplication could lead to vulnerabilities, such as URL tampering.	Ensure secure URL signature generation and validation, and apply <code>ValidateSignature</code> only to routes that need signed URLs.

3. Recommendations and Conclusion

The application demonstrates medium-risk vulnerabilities that require immediate attention. The following measures should be prioritized:

5. **Security Headers:** Configure HSTS, CSP, and other security headers to mitigate XSS, MITM, and MIME sniffing attacks.
6. **Input Validation:** Enforce strict validation for all user inputs to prevent SQL injection, XSS, and other injection-based attacks.
7. **Secure Cookies:** Mark all session and CSRF-related cookies with the `HttpOnly` and `Secure` flags.
8. **Periodic Scanning:** Regularly conduct security scans using tools like OWASP ZAP, Nikto, and SQLMap to identify and fix vulnerabilities proactively.

Addressing these issues will significantly enhance the security posture of the application and protect user data from potential attacks. Further testing and audits are recommended after implementing these changes to ensure all vulnerabilities are adequately addressed.