Penetration Test Report

Prepared for: Redacted Inc. | **Performed By:** EVA | **Date:** June 1, 2025

TABLE OF CONTENTS

Executive Summary

Engagement Overview

[CLIENT NAME] engaged EVA to perform a web application penetration test on the dev.[DOMAIN].ai frontend and api.dev.[DOMAIN].ai backend API. This report contains the findings resulting from that work and a summary of the actions taken by EVA during the engagement.

This is an in-depth assessment of a web application in order to find vulnerabilities that arise from programming errors or flaws in business logic. EVA used both programmatic interaction and automated logic to identify vulnerabilities.

Key Findings & Deficiencies

The findings outlined in this section are significant issues that ultimately result from the technical deficiencies discussed in the body of this report. Addressing the following items will strengthen the overall security posture of [CLIENT NAME] and assist in remediating the most serious issues discovered during this assessment. For additional details and the full list of findings, see the Testing Results Summary below.

- **Critical Privilege Escalation**: An authenticated user (User A, ID 2), initially with "EDITOR" privileges, was able to escalate their privileges to full "ADMIN" by directly modifying their user profile's role attribute via an API request.

- **Excessive Administrative Permissions & Data Exposure**: Once administrative privileges were obtained, the ADMIN user demonstrated the ability to list all users, view any user's full profile data (including email, phone number, name, company), modify any user's profile data (including changing their role), and delete other user accounts. This constitutes a severe breach of data segregation and access control.

- **Stored Cross-Site Scripting (XSS) Risk**: The application allows raw HTML and script payloads to be stored in multiple user and creator profile fields via API interactions. While direct execution was not observed in the UI views tested by the AI agent (likely due to frontend framework sanitization), the storage of unsanitized input presents a significant risk if this data is rendered unescaped in any other context or future UI component.

- **Potential Use of Components with Known Vulnerabilities**: Specific versions of key technologies (Next.js, React, Gunicorn, Auth.js) were largely unobtainable during the assessment. Operating without knowledge of component versions makes it difficult to proactively address known vulnerabilities.

- **Lack of Comprehensive Rate Limiting**: Testing revealed an absence of effective rate limiting on critical API endpoints, including login attempts (both successful and failed) and resource creation/modification, at the volumes tested.

Positive Findings & Strengths

This section is here to help decision makers maintain a broader perspective while planning how to allocate resources to address the results of this assessment.

- **Robust JWT Security Fundamentals**: The application employs JWTs (HS256) for session management. Key security aspects were found to be well-implemented:

- The alg:none JWT vulnerability is not present; such tokens are rejected.

- Token expiration is enforced.

- Logout functionality (both UI-initiated and via POST /api/auth/logout/) effectively invalidates the JWT on the server-side.

- **Strong Input Validation for Data Formats & Choices**: Many API endpoints demonstrate good validation for expected data formats (e.g., email, phone numbers), data types, field length restrictions, and choices for enumerated fields (e.g., user roles, creator main_role).

- **Protection Against Admin Self-Deletion**: The API correctly prevents an administrator from deleting their own account.

- **Secure CORS Policy**: The API's Cross-Origin Resource Sharing policy is configured securely, restricting access to the intended frontend domain (https://dev.[DOMAIN].ai).

- **Effective Handling of Non-Existent Resources**: API requests for non-existent resources (e.g., users, creators) consistently return appropriate 404 Not Found errors.

Risk Analysis

Information Security is founded upon the science and discipline of Risk Management: it is NOT about risk elimination. [CLIENT NAME] is responsible for making the strategic and day-to-day tactical decisions that weigh the security risks of any potential threat against the costs of countermeasures and the organization's ability to achieve its business objectives.

Based on the key findings in this report, EVA considers the overall risk to [CLIENT NAME] to be **Critical**.

REASON FOR RATING: The ability for an authenticated user to escalate their privileges from "EDITOR" to "ADMIN" is a critical flaw. This elevated access then allowed the user to view, modify, and delete any other user's data and account within the system, representing a complete compromise of user data integrity and confidentiality available via the API.

It is important to note that this report represents a snapshot of the security of the environment assessed at a specific point in time. Conditions may have improved, deteriorated, or remained the same since this assessment was completed.

EVA cannot guarantee it will find, locate, discover, and/or repair any or all of [CLIENT NAME]'s system vulnerabilities, breaches, or attempted breaches.

Strategic Guidance

This section contains strategic and tactical goals for [CLIENT NAME] to achieve on the way to remediating the deficiencies identified by this assessment. The Findings section of this report provides more context and more detailed remediation advice for each individual finding.

- **Prioritize Remediation of Privilege Escalation**: Immediately address the vulnerability allowing users to modify their own role attribute. Implement strict server-side validation to ensure only authorized administrative processes can alter user roles.

- **Enforce Principle of Least Privilege for Administrative Roles**: Review the permissions granted to ADMIN users. While admins require broad access, ensure that actions like modifying arbitrary user data or roles are logged, audited, and potentially require specific admin interface actions rather than direct API manipulation where feasible.

- **Implement Comprehensive Output Encoding**: While frontend sanitization appears to be mitigating some XSS risks currently, ensure robust, context-aware output encoding is applied universally wherever user-supplied data is rendered to prevent Stored XSS. Do not rely solely on frontend frameworks.

- **Establish Component Version Management & Patching**: Implement a system for tracking versions of all third-party components (frontend and backend) and a policy for timely application of security patches.

- **Implement Robust Rate Limiting**: Deploy rate-limiting mechanisms on authentication endpoints (for both successful and failed attempts) and sensitive or resource-intensive API operations to protect against brute-force attacks and denial-of-service.

- **Enhance Security Headers**: Deploy missing HTTP security headers like Strict-Transport-Security and Content-Security-Policy on both API and frontend responses.

- **Resolve Application Blockers**: Investigate and resolve the contract template loading error, as this is a major blocker to application functionality and further security testing of that module.

- **Review API Design for User Creation**: The inability for even an administrator to create users via the /api/users/ endpoint suggests user creation occurs via a different, undiscovered mechanism or is handled externally. This should be understood and secured.

EVA sincerely appreciates the opportunity to have worked with [CLIENT NAME] on this engagement. Should you have any questions regarding these findings or the contents of this report, please feel free to contact your designated point of contact.

The following section, Testing Results Summary, contains the complete list of findings discovered during this assessment with background information and remediation recommendations. In addition, a detailed description of the testing methodology is included in Appendix A.

Testing Results Summary

This section includes a summary list that names each finding and its associated risk level. The risk level is assigned according to the guidelines below. Following the summary is a full description of each finding along with remediation recommendations.

Risk Categories

Critical - (C): Vulnerabilities that are being actively exploited in the wild by a worm, other malware, or publicly available exploit leading to remote code execution or providing attackers with a foothold in the environment. This class of vulnerability also includes hosts found already compromised and certain cases where sensitive data was exposed. Immediate action is required.

High - (H): Vulnerabilities that are known to lead to remote exploitation. Certain areas may require immediate attention in the form of a workaround or temporary protection from the network until a sustainable long-term solution can be completed.

Medium - (M): Vulnerabilities or services revealing information or providing functionality that could lead to a system becoming compromised. This class of vulnerabilities may require preconditions or a level of effort for the attacker that makes exploitation less likely. It also includes some vulnerabilities that give attackers information that could be useful in further attacks.

Low - (L): Vulnerabilities or services that could lead to information leakage. This class of vulnerabilities may also be exploitable, but mostly includes situations that give attackers information to launch further attacks.

Informational - (I): Either a potentially risky condition that could not be confirmed by the tester, or a situation that warrants attention but may not require immediate action.

Findings Summary

WEB APPLICATION PENETRATION TEST FINDINGS

CRITICAL-01 .... Vertical Privilege Escalation & Excessive Admin Permissions
MEDIUM-01 .... Stored Cross-Site Scripting (XSS) Risk MEDIUM-02 ....
Potential Use of Components with Known Vulnerabilities INFO-01 .... Missing
Security Headers INFO-02 .... Lack of Comprehensive Rate Limiting

Web Application Penetration Test Findings

CRITICAL-01 Vertical Privilege Escalation & Excessive Admin Permissions

Observation:

An authenticated user (User A, ID 2), initially assigned the "EDITOR" role, was able to escalate their privileges to "ADMIN" by sending a PATCH request to their own user profile endpoint (/api/users/2/) with a modified role attribute in the JSON payload (e.g., {"role": "ADMIN"}). This change was persisted in the database. Upon re-authentication (logout and login), User A's session and JWT reflected the "ADMIN" role.

With these elevated "ADMIN" privileges, User A demonstrated the following capabilities via API interactions:

- List all users in the system (GET /api/users/).
- View the complete profile data of any other user by their ID (GET /api/users/{other_user_id}/).
- Modify any attribute of any other user's profile, including changing their role to "ADMIN" or any other value, updating their email, first_name, company_name, etc. (PATCH /api/users/{other_user_id}/).
- Delete any other user account from the system (DELETE /api/users/{other_user_id}/).

These actions effectively grant a compromised "EDITOR" account (or any account that can modify its own role) complete control over all user accounts and their associated data within the application accessible via the API.

Screenshots/Evidence (Conceptual based on log):

- Log entry showing PATCH /api/users/2/ with {"role": "ADMIN"} returning 200 OK.
- Log entry showing User A's session after re-login, with role as "ADMIN".
- Log entry showing GET /api/users/ by admin User A successfully listing multiple users (if User B existed at that point or if default users are present).
- Log entry showing PATCH /api/users/3/ (User B) by admin User A successfully changing User B's role or email.
- Log entry showing DELETE /api/users/3/ (User B) by admin User A returning 204 No Content.

Discussion:

This represents a critical breakdown in access control and authorization. The role attribute, being a highly sensitive security parameter, should not be modifiable by non-administrative users, especially on their own profile. Once administrative privileges are gained, the subsequent ability to arbitrarily access and manipulate all other user data without restriction signifies excessive permissions or lack of proper authorization checks even for admin actions. This could lead to unauthorized data disclosure, modification, deletion, and full account takeover of any user in the system.

Recommendations:

- **Protect User Role Modification**: Ensure that the role (and groups) attribute in user profiles can only be modified by a highly privileged administrative process or specific, authorized administrative accounts, and never directly by the user themselves, regardless of their current role. Implement strict server-side validation to enforce this.

- **Enforce Principle of Least Privilege for Admins**: While administrators require elevated access, review and restrict the scope of what an admin can do. For instance, modifying critical attributes of other users (like their role or email) should ideally be distinct, audited administrative functions, perhaps available only through a specific admin interface rather than general API endpoints, if business logic allows.

- **Implement Granular Authorization Checks**: For every API request that accesses or modifies data, especially user data, verify that the authenticated user (even if an admin) has the explicit permission to perform that specific action on that specific resource.

- **Audit Logging**: Implement comprehensive audit logging for all administrative actions and sensitive data modifications, especially changes to user roles and profiles.

References:

- OWASP Top Ten: A01:2021-Broken Access Control
- OWASP Testing Guide: Testing for Role and Privilege Manipulation (OTG-AUTHZ-003)

MEDIUM-01 Stored Cross-Site Scripting (XSS)

Observation:

The application API allows for the storage of raw HTML and JavaScript payloads in several user-updatable fields. EVA was able to successfully submit and retrieve payloads like `<script>alert('XSS')</script>` and `<img src=x onerror=alert('XSS_Img')>` in the following fields via PATCH requests:

User Profile (/api/users/{id}/): first_name, last_name, company_name.

Creator Profile (/api/creator/{id}/): name, details, address.

Subsequent GET requests to the respective API endpoints confirmed that these payloads were stored unmodified in the database and returned as part of the JSON response.

During UI testing with Playwright, when these fields were observed (e.g., company_name and first_name in the /settings page textbox, creator name and details on /app/creators and /app/creators/{id} pages), the script tags and HTML were rendered as literal text and did not execute. This suggests that the Next.js/React frontend is likely performing default sanitization or escaping when rendering these specific values in these specific UI components.

Screenshots/Evidence (Conceptual based on log):

- Log entry showing PATCH /api/users/2/ with {"company_name": "[XSS_PAYLOAD]"} returning 200 OK.
- Log entry showing subsequent GET /api/users/2/ response with the raw XSS payload in company_name.
- Log entry showing Playwright navigating to /settings and the snapshot indicating the script tags are visible as text in the input field.
- Similar log entries for creator fields.

Discussion:

Stored Cross-Site Scripting (XSS) occurs when an application stores user-supplied data without proper sanitization and then renders that data back to other users (or the same user) in a web page. Even though direct execution was not observed in the specific UI views tested by EVA, the fact that the API stores raw HTML/script payloads is a significant concern.

The current lack of execution relies on the frontend framework's default behavior in the tested components. If this data is ever:

- Rendered in a different UI component that does not sanitize it.
- Used in a different context (e.g., email notifications, reports).
- Accessed by a client application that processes the API JSON and renders it as HTML without its own sanitization.

Then the stored XSS payloads could execute, leading to session hijacking, data theft, phishing, or other malicious actions within the context of the victim user's browser.

Recommendations:

- **Primary: Implement Context-Aware Output Encoding**: The most robust defense against XSS is to ensure that all user-supplied data is properly encoded or escaped immediately before it is rendered in an HTML context, appropriate to that context (e.g., HTML entity encoding for HTML body, JavaScript escaping for script blocks, attribute escaping for HTML attributes). Do not rely solely on frontend framework defaults; be explicit.

- **Secondary: Input Validation/Sanitization on Storage**: While output encoding is primary, consider implementing input validation or sanitization on the server-side upon storage as a defense-in-depth measure. This could involve stripping known dangerous HTML tags or using a library to clean the HTML. However, this can be complex and may break legitimate user input if not done carefully.

- **Content Security Policy (CSP)**: Implement a strong CSP (see INFO-01) to further restrict the capabilities of any XSS payload that might bypass other defenses.

References:

- OWASP Top Ten: A03:2021-Injection (XSS is a type of injection)
- OWASP Cross-Site Scripting (XSS)
- OWASP XSS Prevention Cheat Sheet

MEDIUM-02 Potential Use of Components with Known Vulnerabilities

Observation:

EVA identified the following technologies in use:

- Frontend: Next.js, React

- Backend API: Gunicorn (Python)
- Authentication: Auth.js (likely NextAuth.js)

Despite multiple attempts to retrieve specific version numbers for these components (e.g., by accessing package.json, Next.js build files, or looking for version information in JavaScript variables or HTTP headers), definitive version information was largely unobtainable. The AI log repeatedly notes "Client-side JavaScript library names and versions for CVE research" as critically awaited user input.

Screenshots/Evidence (Conceptual based on log):

- Log entries showing failed attempts to GET /package.json, /_next/static/build-manifest.json, etc.
- Log entries where EVA requests version information from the user.

Discussion:

Web applications are commonly built using third-party libraries and frameworks. When security vulnerabilities are discovered in these components, they are often publicly disclosed, along with exploits. Attackers actively scan for applications using outdated components with known vulnerabilities.

Without knowing the specific versions of Next.js, React, Gunicorn, Auth.js, and other underlying libraries, it is impossible to determine if [CLIENT NAME] is exposed to known vulnerabilities affecting these components. If outdated versions are in use, the application could be at risk of exploitation. This is item A06:2021-Vulnerable and Outdated Components on the OWASP Top Ten.

Recommendations:

- **Establish Component Inventory and Version Tracking**: Implement a robust process to identify all third-party components and their versions used in both the frontend and backend of the application. This can be achieved through:

- Reviewing package.json, requirements.txt, or other dependency management files.

- Using software composition analysis (SCA) tools.

- Ensuring build processes make version information accessible (e.g., in a build manifest or specific endpoint for privileged users).

- **Implement a Patch Management Policy**: Establish and enforce a policy for regularly checking for security updates for all third-party components and applying patches within a defined, acceptable timeframe.

- **Subscribe to Security Advisories**: Monitor security mailing lists, vendor announcements, and vulnerability databases (e.g., NVD, Snyk, GitHub Advisories) for disclosures related to the components in use.

References:

- OWASP Top Ten: A06:2021-Vulnerable and Outdated Components
- NIST National Vulnerability Database (NVD): https://nvd.nist.gov/

INFO-01 Missing Security Headers

Observation:

EVA evaluated the HTTP security headers returned by both the API (api.dev.[DOMAIN].ai) and the frontend application (dev.[DOMAIN].ai).

API (api.dev.[DOMAIN].ai):

- Present: X-Frame-Options: DENY, X-Content-Type-Options: nosniff, Referrer-Policy: same-origin, Cross-Origin-Opener-Policy: same-origin.
- Missing: Strict-Transport-Security (HSTS), Content-Security-Policy (CSP), Permissions-Policy.

Frontend (dev.[DOMAIN].ai):

Initial checks in the log noted the frontend was missing HSTS, CSP, X-Content-Type-Options, and X-Frame-Options. A full header list from the user for a specific frontend page (/app/creators) was still pending at the conclusion of the summarized logs, so a complete frontend assessment is not yet finalized.

Screenshots/Evidence (Conceptual based on log):

- Log entries detailing headers observed from curl requests to API endpoints.
- Log entries noting missing headers for dev.[DOMAIN].ai and the pending request for full frontend headers.

Discussion:

HTTP security headers provide an additional layer of defense by instructing the browser on how to behave when handling the application's content, mitigating risks like clickjacking, cross-site scripting, and man-in-the-middle attacks.

- **Strict-Transport-Security (HSTS)**: Enforces the use of HTTPS, protecting against protocol downgrade attacks and cookie hijacking.
- **Content-Security-Policy (CSP)**: Helps prevent XSS by defining allowed sources for content (scripts, styles, images, etc.).
- **Permissions-Policy (formerly Feature-Policy)**: Allows control over which browser features (e.g., camera, microphone, geolocation) the application can use.
- **X-Frame-Options**: Protects against clickjacking by controlling whether the site can be embedded in an iframe. (Present on API, status for frontend needs confirmation).
- **X-Content-Type-Options**: Prevents browsers from MIME-sniffing a response away from the declared content type. (Present on API, status for frontend needs confirmation).

Recommendations:

- **Implement Strict-Transport-Security (HSTS)**: For both API and frontend, deploy the HSTS header (e.g., Strict-Transport-Security: max-age=31536000; includeSubDomains; preload) once confident that all subdomains support HTTPS. Consider submitting the domain for HSTS preloading.

- **Implement Content-Security-Policy (CSP)**: Develop and deploy a restrictive CSP for both the API (can be simpler, e.g., default-src 'none') and especially the frontend application. Start with a report-only mode to identify necessary directives before enforcing.

- **Implement Permissions-Policy**: Define a Permissions-Policy header specifying only the browser features the application legitimately requires.

- **Confirm Frontend Headers**: Once the full list of headers for dev.[DOMAIN].ai is available, ensure X-Frame-Options: DENY (or SAMEORIGIN) and X-Content-Type-Options: nosniff are present if found missing.

References:

- OWASP Secure Headers Project: https://owasp.org/www-project-secure-headers/
- MDN Web Docs for each respective header (e.g., Strict-Transport-Security, Content-Security-Policy).

INFO-02 Lack of Comprehensive Rate Limiting

Observation:

EVA performed tests to assess rate limiting on several API endpoints:

Login Endpoint (POST /api/auth/login/):

- 30 successful login attempts for User A within 10-12 seconds were all processed with 200 OK responses, each issuing a new token.
- 30 failed login attempts (valid user, invalid password) within 10-12 seconds all returned 400 Bad Request without any blocking, lockout, or 429 responses.

Resource Creation (POST /api/creator/): 30 requests to create creators within ~10 seconds all succeeded with 201 Created.

Resource Modification (PATCH /api/users/2/): 30 requests to update User A's profile within ~8 seconds all succeeded with 200 OK.

Authenticated GET Endpoints (GET /api/auth/session, GET /api/users/2/): Bursts of 10-20 requests were processed without issue.

The primary form login endpoint (/api/auth/callback/credentials) appeared to require a different flow (likely CSRF token) and was not easily testable for rate limiting with simple JSON POSTs by the AI.

Discussion:

The absence of effective rate limiting on authentication endpoints makes the application susceptible to brute-force attacks against user credentials and credential stuffing attacks. While a strong password policy can mitigate this, rate limiting provides an essential layer of defense.

Lack of rate limiting on resource creation and modification endpoints can lead to:

- Denial of Service (DoS) by overwhelming server resources or filling up storage.
- API abuse, such as rapidly creating spam content or performing bulk modifications.
- Making other enumeration or brute-force attacks against other parameters more feasible.

Recommendations:

- **Implement Rate Limiting on Authentication**:

- Apply strict rate limits to POST /api/auth/login/ for both successful and failed attempts based on source IP address and/or username. Consider implementing temporary account lockouts or CAPTCHA challenges after a certain number of failed attempts.

- Ensure the primary form-based login mechanism (/api/auth/callback/credentials) also has robust rate limiting and anti-brute-force protections.

- **Implement Rate Limiting on Resource-Intensive API Endpoints**: Apply rate limits to endpoints for creating (POST), modifying (PATCH), and potentially listing (GET if expensive) resources like users and creators. Limits should be based on factors like source IP and authenticated user ID.

- **Logging and Monitoring**: Implement logging and alerting for excessive requests or repeated failed authentication attempts to detect and respond to potential attacks.

References:

- OWASP Top Ten: A07:2021-Identification and Authentication Failures (relevant to brute-force on login)
- OWASP API Security Top 10: API4:2019-Lack of Resources & Rate Limiting

Appendix A: Methodology

Web Application Penetration Test

The target for this portion of the assessment was the web application hosted at dev.[DOMAIN].ai (frontend) and the backend API at api.dev.[DOMAIN].ai. EVA, an AI penetration testing agent, performed the assessment.

The methodology involved several phases, driven by EVA's internal logic and programmatic interaction with the target:

Reconnaissance and Information Gathering:

- Identifying target domains and key API endpoints.
- Attempting to fingerprint technologies (Next.js, Gunicorn, Auth.js) and their versions.
- Understanding the authentication mechanism (JWTs, session cookies).

Authentication and Session Management Testing:

- Analyzing JWT structure, algorithm (HS256), and lifetime.
- Testing token validity, expiration, and invalidation upon logout.
- Probing for alg:none and other common JWT vulnerabilities.
- Examining password reset functionality (not found).

Application Mapping and API Endpoint Discovery:

- Using initial target endpoints and common API path patterns to discover accessible functionalities.
- Employing OPTIONS requests to determine allowed HTTP methods for identified endpoints.
- Attempting to find API documentation (e.g., OpenAPI/Swagger files).

Automated and Manual Vulnerability Probing (Emulated):

EVA used curl commands (via term_run_command) for direct API interaction and Playwright for browser interaction (login, UI observation).

Input Validation and Injection Testing: Systematically testing input fields (URL parameters, JSON body parameters) for common injection vulnerabilities:

- Cross-Site Scripting (XSS): Stored and Reflected (API-focused).
- SQL Injection (SQLi): Basic and advanced probes.
- Server-Side Template Injection (SSTI): Probes for common Python template engines.
- OS Command Injection: Probes using shell metacharacters.

Authorization and Access Control Testing:

- Testing for Insecure Direct Object References (IDOR) and Broken Access Control (BAC), initially focused on User A's capabilities after privilege escalation.
- Testing for Vertical Privilege Escalation (User A modifying own role).
- (Extensive authorization testing involving User B was planned for the next phase, following receipt of User B's credentials at the end of the summarized log period).

Business Logic Flaw Identification: Examining API behavior for unexpected outcomes, especially around user and creator management.

Error Handling Analysis: Probing for verbose error messages or information leakage.

Rate Limiting Tests: Sending bursts of requests to various endpoints.

Specific Vulnerability Checks:

- Server-Side Request Forgery (SSRF): Probing string parameters for URL interpretation.
- HTTP Parameter Pollution (HPP): Testing how duplicated parameters are handled.
- Insecure Deserialization: Conceptual consideration, though no direct vectors identified in the JSON API.
- File Upload Vulnerabilities: Attempting to identify upload functionalities (none found explicitly via API).

The AI agent's process is iterative. Findings from one phase (e.g., successful privilege escalation) directly inform the strategy and targets for subsequent phases (e.g., testing admin capabilities). Blockers, such as the inability to create contracts or lack of a second user account for parts of the test, were noted, and testing was adapted or deferred accordingly.

Security Headers and SSL Checks

EVA reviewed API responses for security headers. The frontend application (dev.[DOMAIN].ai) has been noted as potentially missing several headers, but a full analysis awaits user-provided data. SSL/TLS configuration details (e.g., from Qualys SSL Labs) were not explicitly mentioned in the AI's log output provided, so this aspect cannot be fully reported here beyond the API header checks.

API (api.dev.[DOMAIN].ai) Security Headers Observed:

- X-Frame-Options: DENY
- X-Content-Type-Options: nosniff
- Referrer-Policy: same-origin
- Cross-Origin-Opener-Policy: same-origin

Missing from API (Notable):

- Strict-Transport-Security (HSTS)
- Content-Security-Policy (CSP)
- Permissions-Policy

The finding, INFO-01 Missing Security Headers, in the Testing Results Summary section includes further discussion and remediation recommendations for this issue.

Third-Party Content

EVA attempted to identify technologies used by the application and their versions. The following technologies were identified:

- Frontend: Next.js, React
- Backend API: Gunicorn (Python)
- Authentication: Auth.js (likely NextAuth.js)
- CDN: CloudFront (noted in initial summary, less focused on by AI later)

Specific version numbers for these components were largely unobtainable despite various attempts by EVA (e.g., checking for package.json, Next.js build files, common version variables). The API root (GET /api/) did return a body indicating "version":"1.0.0", but this appears to be an internal API version rather than a specific framework version.

The finding, MEDIUM-02 Potential Use of Components with Known Vulnerabilities, in the Testing Results Summary section includes further discussion and remediation recommendations for this issue due to the unknown versions.

Example of how Wappalyzer output might look if it were available visually:

- JavaScript Frameworks: Next.js, React
- Web Servers: Gunicorn
- Programming Language: Python
- Authentication: Auth.js
- CDN: Amazon CloudFront

Authentication and Session Management

EVA investigated the application's authentication and session management:

Authentication Mechanism: The application uses Auth.js (likely NextAuth.js) for authentication.

Tokens: JSON Web Tokens (JWTs) are issued.

- Algorithm: HS256.
- Access Token Lifetime: 30 minutes.
- Refresh Token Lifetime: 14 days (mentioned in initial summary, less directly tested by AI later).
- Structure: Standard JWT structure (header, payload, signature). Header contains alg: HS256, typ: JWT. Payload contains exp, iat, jti, user_id, and token_type.

Session Cookies: The primary session token appears to be __Secure-authjs.session-token (HttpOnly, Secure).

Security of Tokens:

- alg:none vulnerability was tested and is NOT present; tokens with alg:none are rejected.
- Signature validation appears to be in place (tampering with payload and using original signature results in token rejection).
- Token expiration is enforced; expired tokens are rejected.

Logout:

- UI-initiated logout works.
- A POST /api/auth/logout/ endpoint was identified, which successfully logs out the user and invalidates the token.

Password Reset: Functionality for password reset was not found during EVA's exploration.

Username Enumeration: Not explicitly tested for in the provided logs, but no findings related to it were reported.

Privilege Persistence in Tokens: It was observed that if a user's role is changed in the database (e.g., ADMIN to "viewer"), an existing valid JWT for that user retains its original privileges until it expires or is invalidated by logout. New tokens issued after re-authentication correctly reflect the updated role from the database.

CSRF Protection: An Auth.js CSRF cookie was noted as present. An attempt to test SSRF via Auth.js localhost callback URLs was speculative and inconclusive without an OAST tool.

Overall, many aspects of authentication and session management (especially JWT handling) appear robust. The key concern is the persistence of privileges in active tokens after a backend role change.

Mapping the Application

EVA explored the application by interacting with known and guessed API endpoints. Key identified API paths include:

- /api/auth/session (GET for session data, POST for password update - though this use for password update needs more clarity)
- /api/auth/providers
- /api/auth/error
- /api/auth/login/ (POST for login)
- /api/auth/logout/ (POST for logout)
- /api/users/ (GET for user list - admin only)
- /api/users/{user_id}/ (GET, PATCH, DELETE for specific user)
- /api/creator/ (GET for creator list - unclear if all or user-specific for non-admin, POST for creation)
- /api/creator/{creator_id}/ (GET, PATCH, DELETE for specific creator)
- /api/contracts/ (GET for contract list, POST for creation - blocked)
- /api/contracts/template/{TEMPLATE_NAME} (GET for template - blocked)

The API root (/api/) returns a welcome message and API version "1.0.0". No OpenAPI/Swagger specification files were found at common locations.

Scanning, Fuzzing, and Injection

EVA performed various fuzzing and injection tests against the identified API endpoints and parameters:

SQL Injection: Basic SQLi payloads (single quotes, OR '1'='1', time-based stubs like pg_sleep(5)) were injected into user and creator profile string fields via PATCH requests, and into discovered GET query parameters. In all tested cases, the payloads were either stored literally (for PATCHed fields) or did not cause errors/delays, suggesting protection (likely via

ORM or parameterized queries). Path parameters like user IDs also appeared type-checked, preventing SQLi.

Server-Side Template Injection (SSTI): Basic SSTI payloads (e.g., {{77}}, ${77}}, attempts to access {{config}} or simple Python evaluations) were injected into user and creator profile string fields. These were consistently stored literally, with no evidence of server-side template execution.

OS Command Injection: Basic OS command injection payloads (e.g., | id, ; id, $(whoami)) injected into user and creator profile string fields were stored literally. The template_name parameter for contract creation (a potential vector) remains untestable due to the template loading blocker.

Error Handling: Tests involving malformed JSON, incorrect data types, overly long strings, and special characters generally resulted in standard validation errors (400 Bad Request) or generic server errors (e.g., 404, 405, 415) without leaking verbose internal details, stack traces, or sensitive paths.

Cross-Site Scripting (XSS)

As detailed in finding MEDIUM-01, EVA found that several user and creator profile fields allow raw HTML and JavaScript payloads (e.g., `<script>alert('XSS')</script>`, `<img src=x onerror=alert('XSS_Img')>`) to be stored via API PATCH requests. These payloads were confirmed to be returned unmodified in subsequent API GET responses.

However, when EVA used Playwright to view these fields in the application's UI (e.g., /settings, /app/creators, /app/creators/{id}), the scripts were rendered as inert text, likely due to default sanitization/escaping by the Next.js/React frontend framework in those specific components. Direct execution was not observed in these views.

Fields found to store XSS payloads:

- User: first_name, last_name, company_name.
- Creator: name, details, address.

Fields that rejected XSS payloads due to format validation: User phone_number; Creator main_role, email, phone_number.

Insecure Direct Object Reference & Authorization Bypass

This section covers observations related to accessing or modifying resources without proper authorization, primarily focusing on the actions possible after User A escalated privileges to ADMIN. Testing from the perspective of a non-admin user (User B) was planned for the next wave.

Vertical Privilege Escalation (User A):

User A (ID 2), initially an "EDITOR", successfully changed their own role to "ADMIN" via PATCH /api/users/2/. This is the most critical authorization bypass identified (see CRITICAL-01).

Admin Access to Other Users' Data (Post-Escalation):

- As ADMIN, User A could list all users (GET /api/users/).
- As ADMIN, User A could view the full profile of any other user (GET /api/users/{other_user_id}/).
- As ADMIN, User A could modify any attribute of any other user, including their role, effectively allowing an admin to promote/demote others or take over accounts (PATCH /api/users/{other_user_id}/).
- As ADMIN, User A could delete any other user (DELETE /api/users/{other_user_id}/).

Admin Access to Creators:

- As ADMIN, User A could create creators (POST /api/creator/), and these were correctly assigned user_id: 2.
- As ADMIN, User A could list, view, update, and delete their own creators.
- Testing admin access to creators owned by other users is pending the creation of User B and User B creating their own resources.

Creator Ownership Parameters:

- When creating a creator, if user_id is omitted, it defaults to the authenticated user (User A, ID 2).
- Attempting to create a creator with user_id: null results in a 400 Bad Request ("This field may not be null.").
- Attempting to create a creator with user_id set to a non-existent user ID results in a 400 Bad Request ("Invalid pk... object does not exist.").
- Attempting to PATCH an existing creator's user_id to a non-existent user ID also results in a 400 Bad Request.
- The ability for an admin to change an existing creator's user_id to another valid existing user (reassign ownership) is pending the availability of User B.

Self-Modification Restrictions (Even for Admin):

- Admin User A could change their own role between "ADMIN", "EDITOR", and "viewer". Invalid roles were rejected.
- Admin User A could NOT modify their own groups array via PATCH /api/users/2/; the field was ignored.
- Admin User A could NOT modify their own id via PATCH /api/users/2/.

- Admin User A could NOT delete their own account (DELETE /api/users/2/ resulted in 403 Forbidden "Deleting your own account is not allowed.").

Access to Non-Existent Resources:

Requests (GET, PATCH, DELETE) to /api/users/{non_existent_id}/ or /api/creator/{non_existent_id}/ correctly resulted in 404 Not Found errors.

The core issue here is the initial privilege escalation. Once admin rights are obtained, the API grants very broad, largely unrestricted access to manage all user and creator data. Fine-grained authorization checks for specific administrative actions beyond basic role checking appear limited.

Appendix B: Points of Contact and Scope Listing

The primary point of contact for this assessment was: **[REDACTED USER CONTACT INFORMATION]**

Web Application Penetration Test Scope

- **Frontend Application**: https://dev.[DOMAIN].ai
- **Backend API**: https://api.dev.[DOMAIN].ai
- **User accounts primarily used for testing**:
- User A: [REDACTED EMAIL] (Password: [REDACTED], User ID: 2) - Role escalated from "EDITOR" to "ADMIN".
- User B: [REDACTED EMAIL] (Password: [REDACTED], User ID: 3) - Role: "EDITOR". (Note: Testing from User B's perspective was planned to commence after the summarized log period)