The Whirling Valley of Crooked Fools

Reynard Security Research Team
../docs/research/optimus/favicon.pdf

September 7, 2025

Abstract

This paper details an automated security assessment conducted on the Reynard backend system. The objective is to identify and document potential security vulnerabilities through various attack simulations. The findings will be used to strengthen the backend's resilience against malicious activities.

1 Introduction

The Reynard application aims to provide a robust platform for image management and caption generation. As with any web application, the security of its backend infrastructure is paramount. This research paper outlines the methodology, tools, and findings of a penetration testing exercise performed on the Reynard backend. The primary goal is to proactively identify and address weaknesses before they can be exploited by external threats. Penetration testing is a crucial practice in identifying vulnerabilities and enhancing system security [3, 2], and its principles can be applied to various scales of systems, from individual applications to complex urban infrastructures [1].

The backend is a Python-based application, serving various functionalities including file operations, metadata management, and caption generation. The frontend interacts with the backend via a well-defined API. This assessment focuses on potential vulnerabilities within this API and the underlying backend services.

2 Methodology

Our approach involves a series of automated tests designed to simulate common attack vectors. These include, but are not limited to, injection attacks, authentication bypasses, authorization flaws, and denial-of-service attempts. Python scripts will be developed to execute these tests, providing a systematic and reproducible way to probe for weaknesses. This methodology is inspired by established penetration testing frameworks [4], and aligns with principles discussed in broader security assessments [3].

3 Threat Model and Scope Definition

Our threat model focuses on unauthenticated and authenticated attackers attempting to compromise the Reynard backend system. The scope of this assessment is primarily limited to the backend API endpoints and their interactions with the database and file system. We specifically target vulnerabilities aligned with the OWASP Top 10 risks, including:

- **Broken Access Control:** Ensuring proper authentication and authorization mechanisms are in place and cannot be bypassed.
- **Cryptographic Failures:** Assessing how sensitive data is handled, stored, and transmitted.
- **Injection:** Probing for SQL Injection, Command Injection, and Path Traversal vulnerabilities.
- **Insecure Design:** Identifying design flaws that could lead to vulnerabilities.
- **Security Misconfiguration:** Checking for default configurations, unnecessary features, or improper permissions.
- **Vulnerable and Outdated Components:** Identifying any known vulnerabilities in third-party libraries or frameworks.
- **Identification and Authentication Failures:** Focusing on weaknesses in session management and authentication processes.
- **Software and Data Integrity Failures:** Ensuring that data is not tampered with during transmission or storage.
- **Security Logging and Monitoring Failures:** Assessing the effectiveness of logging and monitoring for security events.
- **Server-Side Request Forgery (SSRF):** Testing if the server can be coerced into making unintended requests to internal or external resources.

Out of scope for this assessment are client-side vulnerabilities like DOM-based XSS (unless they stem from backend reflection), social engineering attacks, and physical security vulnerabilities.

3.1 Environment Setup

The target backend server is running locally with the following configuration:

- **Development Port:** 7000
- **Root Directory:** /home/kade/datasets
- **Node Environment:** development

For improved documentation and reproducibility, we recommend providing containerized (e.g., Docker) or Infrastructure as Code (IaC) scripts to spin up consistent test environments. Additionally, a `requirements.txt` file with pinned dependencies and sample test data should be included to ensure consistent test execution across different environments. The backend process is initiated with the command:

```
DEV_PORT=7000 ROOT_DIR=/home/kade/datasets NODE_ENV=development python -m app
```

Listing 1: Backend Startup Command

4 Initial Reconnaissance

Before launching any attacks, it is crucial to understand the backend's architecture and the API endpoints it exposes. This involves reviewing existing documentation, if any, and observing network traffic between the frontend and backend.

4.1 API Endpoint Discovery

We will begin by identifying the various API endpoints. This can be achieved by analyzing the frontend's source code to see how it interacts with the backend or by observing network requests made by the application during normal operation.

5 Vulnerability Assessment Categories

We will categorize potential vulnerabilities into the following areas:

5.1 Authentication and Authorization

- **Bypass Attacks:** Accessing protected resources without authentication.
- **Privilege Escalation:** Gaining higher-privilege functionalities as a low-privilege user.
- **Insecure Direct Object References (IDOR):** Manipulating user input to access unauthorized resources.

5.2 Input Validation and Injection

- **SQL Injection:** Attempting to inject malicious SQL queries into input fields.
- **Command Injection:** Testing for vulnerabilities that allow execution of arbitrary system commands.
- **Cross-Site Scripting (XSS):** While primarily a frontend vulnerability, we will check if the backend sanitizes user-supplied data correctly before storage or reflection.
- **Path Traversal:** Attempting to access files and directories outside of the intended directory.

5.3 Error Handling and Information Disclosure

- **Verbose Error Messages:** Identifying if error messages reveal sensitive information about the backend infrastructure.
- **Stack Traces:** Checking for exposed stack traces that could aid an attacker.

5.4 Denial of Service (DoS)

- **Resource Exhaustion:** Testing if large or malformed requests can exhaust backend resources.
- **Rate Limiting:** Assessing if the API is vulnerable to brute-force attacks due to missing or inadequate rate limiting.

6 Python Scripting for Exploitation

This section will detail the Python scripts developed to carry out the automated security tests. Each script will target a specific vulnerability category and will be designed to be modular and reusable. To enhance the depth of automated testing, future iterations will incorporate dynamic fuzzers (e.g., OWASP ZAP) and develop tests for chained attacks (e.g., CSRF + IDOR) and complex business logic flaws.

6.1 The Hidden Flag

To facilitate the testing of information disclosure and unauthorized access, a hidden "flag" has been embedded within the backend. This flag is a string value intended to be accessible only through specific, potentially vulnerable API endpoints or misconfigurations. The objective of our security scripts will be to discover and retrieve this flag without authorized means, simulating a real-world scenario where sensitive information might be inadvertently exposed.

Specifically, the flag FLAG{YOU_FOUND_THE_CONFIG_FLAG} has been added to the /api/config endpoint's response. This endpoint is typically used by the frontend to retrieve application configuration. Our tests will attempt to access this endpoint and extract the flag, assessing whether the flag is exposed without proper authentication or authorization, or if any other vulnerability allows its unintended disclosure.

6.2 Setup

All Python scripts will be located in the ./blackhat directory. They will utilize standard Python libraries for making HTTP requests and parsing responses. To improve the rigor of Proof-of-Concept (PoC) scripts and ensure robust false-positive/negative handling, future enhancements will include:

- **Modular Error Handling:** Implementing comprehensive try-except blocks and custom exception handling for unexpected response formats.
- **Retry Logic and Exponential Backoff:** Incorporating retry mechanisms with exponential backoff for transient network issues or rate limiting scenarios.
- **Parameterized Reporting:** Developing a consistent reporting mechanism that captures nuanced failure modes, including detailed status codes, response bodies, and custom flags for test outcomes.
- **Result Validation Pipeline:** Establishing a clear pipeline for triaging false positives and negatives, potentially integrating with a central logging system for automated analysis.

6.3 Authentication Bypass Testing

This section details the tests conducted against the /api/login endpoint to assess its resilience against various authentication bypass techniques, including common SQL injection patterns.

Our login_test.py script was extended to include the following scenarios:

- **Invalid Credentials:** Testing with a non-existent username and incorrect password.
- **Empty Credentials:** Attempting to log in with both username and password fields empty.
- **SQL Injection in Username/Password:** Probing for vulnerabilities by injecting common SQL injection payloads into both username and password fields.
- **Advanced SQL Injection Bypass:** Specifically attempting to bypass authentication using the ’ OR 1=1 - payload in the username field with an empty password.

- **Access to Protected Endpoint After Failed Login:** Verifying that a protected endpoint (`/api/config`) remains inaccessible even after multiple failed login attempts.

6.4 Results: Robust Authentication

Our `login_test.py` script yielded the following results:

```
--- Testing /api/login endpoint ---

[+] Attempting login with invalid credentials...
    Status Code: 401
    Response: {'detail': 'Incorrect username or password'}
    [OK] Login failed as expected for invalid credentials (Status 401 Unauthorized).

[+] Attempting login with empty credentials...
    Status Code: 401
    Response: {'detail': 'Incorrect username or password'}
    [WARNING] Unexpected status code for empty credentials. Expected 400.

[+] Attempting SQL Injection in username field...
    Status Code: 401
    Response: {'detail': 'Incorrect username or password'}
    [OK] SQL Injection attempt in username failed as expected.

[+] Attempting SQL Injection in password field...
    Status Code: 401
    Response: {'detail': 'Incorrect username or password'}
    [OK] SQL Injection attempt in password failed as expected.

[+] Attempting SQL Injection with ' OR 1=1' in username and empty password...
    Status Code: 401
    Response: {'detail': 'Incorrect username or password'}
    [OK] SQL Injection bypass attempt in username with empty password failed as expected.

[+] Attempting to access \texttt{/api/config} without authentication after login attempts...
    Status Code: 401
    Response: {'detail': 'Not authenticated'}
    [OK] Access to \texttt{/api/config} is restricted as expected (authentication/authorization required).

--- /api/login endpoint testing complete ---
```

Listing 2: Output from `login_test.py`

Analysis: All authentication bypass attempts, including basic invalid credentials, SQL injection payloads, and an advanced SQL injection bypass, were successfully thwarted by the backend's authentication mechanism, consistently returning a `401 Unauthorized` status. This indicates a strong defense against these common attack vectors. The `/api/config` endpoint also correctly returned 401 Unauthorized, further confirming the effectiveness of the previously implemented access restrictions.

7 Results and Findings

During the penetration testing, a significant information disclosure vulnerability was identified related to the /api/config endpoint. Our flag_capture_test.py script successfully retrieved a hidden security flag without authentication.

7.1 Vulnerability: Unauthenticated Information Disclosure (/api/config)

- **Endpoint:** /api/config
- **Method:** GET
- **Description:** The /api/config endpoint, previously found to expose sensitive information (the embedded security_flag) to unauthenticated users, has been secured. The endpoint now correctly restricts access, requiring administrative privileges to view the security_flag. This mitigation prevents unauthorized information disclosure.
- **Proof of Concept (flag_capture_test.py output after mitigation):**

```
--- Testing /api/config for security flag ---

[+] Attempting to retrieve flag from /api/config (unauthenticated)...
    Status Code: 401
    [OK] Access to /api/config is restricted as expected (authentication/authorization required).

--- /api/config flag testing complete ---
```

Listing 3: Output from flag_capture_test.py (after mitigation)

- **Impact:** This vulnerability could have led to the exposure of sensitive configuration details. With the implemented fix, the confidentiality of the security_flag is preserved, demonstrating successful mitigation of the information disclosure risk.

7.2 Vulnerability: Unauthenticated Assistant Information Access

- **Endpoints:** /api/ollama/chat (POST), /api/ollama/assistant/tools (GET), and /api/ollama/assistant/context/test_path (GET)

- **Description:** Initial reconnaissance indicated that several assistant-related API endpoints were intended to be protected by user authentication. Our security tests confirmed that these endpoints were indeed secure against unauthenticated access. This is crucial as these endpoints can expose sensitive information (e.g., user chat history, available tools for an authenticated user, or directory context) or allow actions (e.g., chatting with the assistant) that should be restricted to authenticated users.

- **Proof of Concept (`exploit_script.py` output):**

```
--- Testing Unauthenticated Assistant Endpoints ---

[+] Attempting POST request to http://localhost:7000/api/ollama/chat without authentication...
    Status Code: 401
    Response: {'detail': 'Not authenticated'}
    [OK] Access to http://localhost:7000/api/ollama/chat is restricted as expected (Status 401 Unauthorized).

[+] Attempting GET request to http://localhost:7000/api/ollama/assistant/tools without authentication...
    Status Code: 401
    Response: {'detail': 'Not authenticated'}
    [OK] Access to http://localhost:7000/api/ollama/assistant/tools is restricted as expected (Status 401 Unauthorized).

[+] Attempting GET request to http://localhost:7000/api/ollama/assistant/context/test\_path without authentication...
    Status Code: 401
    Response: {'detail': 'Not authenticated'}
    [OK] Access to http://localhost:7000/api/ollama/assistant/context/test\_path is restricted as expected (Status 401 Unauthorized).

--- Unauthenticated Assistant Endpoints Testing Complete ---
```

Listing 4: Output from `exploit_script.py` (Assistant Endpoints)

- **Impact:** The successful restriction of access to these endpoints for unauthenticated users confirms the backend's robust implementation of authentication for assistant-related functionalities. No direct vulnerability was found in these specific endpoints, affirming their secure design.

7.3 Robustness: Login Rate Limiting

- **Endpoint:** `/api/login`

- **Description:** Our `rate_limit_test.py` script was used to assess the effectiveness of the login rate-limiting mechanism. The backend is configured to allow a maximum of 5 login attempts within a 10-minute window before blocking further attempts from the same source. The test script repeatedly sent invalid login requests to simulate a brute-force attack.

- **Proof of Concept (`rate_limit_test.py` output):**

```
--- Testing rate limiting for /api/login ---
Request 1: Status Code: 401, Response: {'detail': 'Incorrect username or password'}
    [POSSIBLE BLOCK] Expected 401 for invalid credentials. If repeated for many requests, it might indicate a block.
Request 2: Status Code: 401, Response: {'detail': 'Incorrect username or password'}
    [POSSIBLE BLOCK] Expected 401 for invalid credentials. If repeated for many requests, it might indicate a block.
Request 3: Status Code: 401, Response: {'detail': 'Incorrect username or password'}
    [POSSIBLE BLOCK] Expected 401 for invalid credentials. If repeated for many requests, it might indicate a block.
Request 4: Status Code: 401, Response: {'detail': 'Incorrect username or password'}
    [POSSIBLE BLOCK] Expected 401 for invalid credentials. If repeated for many requests, it might indicate a block.
Request 5: Status Code: 401, Response: {'detail': 'Incorrect username or password'}
    [POSSIBLE BLOCK] Expected 401 for invalid credentials. If repeated for many requests, it might indicate a block.
Request 6: Status Code: 429, Response: {'detail': 'Too many login attempts. Please try again later.'}
    [BLOCKED] Rate limit hit at request 6 with 429 Too Many Requests.
Request 7: Status Code: 429, Response: {'detail': 'Too many login attempts. Please try again later.'}
    [BLOCKED] Rate limit hit at request 7 with 429 Too Many Requests.
Request 8: Status Code: 429, Response: {'detail': 'Too many login attempts. Please try again later.'}
    [BLOCKED] Rate limit hit at request 8 with 429 Too Many Requests.
Request 9: Status Code: 429, Response: {'detail': 'Too many login attempts. Please try again later.'}
    [BLOCKED] Rate limit hit at request 9 with 429 Too Many Requests.
Request 10: Status Code: 429, Response: {'detail': 'Too many login attempts. Please try again later.'}
    [BLOCKED] Rate limit hit at request 10 with 429 Too Many Requests.
--- Rate limiting test complete. Total requests: 10, Blocked: 10 ---
```

Listing 5: Output from `rate_limit_test.py` (Login Rate Limiting)

- **Analysis:** The test results confirm that the `/api/login` endpoint's rate-limiting mechanism is effectively implemented. After 5 invalid login attempts, subsequent requests from the same source were blocked with a `429 Too Many Requests` status code. This demonstrates strong protection against brute-force login attacks.

7.4 Vulnerability: Backend Configuration Initialization Failure

- **Description:** During application startup, a critical error was identified in the backend's configuration loading process, leading to subsequent failures in other services like Git. Initially, the error message `'Depends' object has no attribute 'role'` was observed. This occurred because the FastAPI endpoint function `get_config`, which relies on dependency injection (`Depends(get_current_user)`), was being called directly during the application's `lifespan` function. In this context, the `Depends` object was not resolved to a `User` instance, causing the attempt to access its `role` attribute to fail.

- **Mitigation - Step 1: Correcting Dependency Injection Context:** To resolve this, the initialization of the `ConfigManager` was moved to occur earlier within the `lifespan` function in `app/main.py`. The application was then configured to load its initial configuration by calling `config_manager.get_config()` instead of the FastAPI endpoint. This ensured that configuration data was retrieved correctly from the initialized `ConfigManager` instance.

```
    global config_manager
    config_manager = initialize_config_manager()
    initial_config = config_manager.get_config()

```

Listing 6: Partial Fix for Config Loading in `app/main.py`

- **Subsequent Issue: `'AppConfig'` object is not subscriptable:** After the initial mitigation, a new `TypeError: 'AppConfig' object is not subscriptable` emerged. This indicated that while `initial_config` was now correctly an `AppConfig` object, subsequent code was still attempting to access its attributes using dictionary-style bracket notation (e.g., `initial_config['thumbnail_size']`) instead of dot notation (e.g., `initial_config.thumbnail_size`).

- **Mitigation - Step 2: Adopting Correct Object Access:** The final fix involved updating all instances where `initial_config` attributes were accessed, particularly in the creation of `CachedFileSystemDataSource` and the setting of thumbnail sizes. All bracket-based accesses were replaced with dot notation, aligning with the `AppConfig` object's structure.

```
    data_source = CachedFileSystemDataSource(
        ROOT_DIR,
        (initial_config.thumbnail_size, initial_config.
thumbnail_size),
        (initial_config.preview_size or initial_config.
thumbnail_size, initial_config.preview_size or initial_
config.thumbnail_size)
    )
    if initial_config.thumbnail_size:
        data_source.set_thumbnail_size((initial_config.thumbnail_size, initial_config.thumbnail_size))

```

Listing 7: Final Fix for AppConfig Access in `app/main.py`

- **Impact:** The complete resolution of these configuration initialization issues ensures that the backend application starts up correctly, allowing dependent services like the Git manager to initialize without error. This stabilizes the backend and allows the frontend to access Git-related functionalities without encountering `'NoneType' object has no attribute 'get_status'` errors.

8 Frontend-Backend Interaction Vulnerabilities

During the development and testing phase, a significant frontend-backend interaction vulnerability was identified, leading to a `401 Unauthorized` error that disrupted the user login flow. This issue was not a direct backend vulnerability, but rather a timing and dependency problem on the frontend's side concerning a protected backend endpoint. To proactively catch these sequencing flaws and ensure robust frontend-backend interaction, we recommend incorporating end-to-end integration tests (e.g., using Playwright) and instrumenting performance metrics for request timing.

8.1 Vulnerability: Premature Unauthenticated `/api/config` Fetch

- **Endpoint:** `/api/config`

- **Description:** The application's frontend was attempting to fetch configuration data from the `/api/config` endpoint before the user was fully authenticated. As `/api/config` is a protected endpoint (as confirmed by the backend security assessment in previous sections), this premature request resulted in a `401 Unauthorized` response, interrupting the login process and displaying an "Session expired. Please log in again." notification.

- **Impact:** This vulnerability directly prevented users from logging into the application, as the essential configuration data could not be retrieved before authentication was established. Although the backend correctly enforced authentication, the frontend's architectural flaw in fetching order created a denial-of-service for login.

8.2 Mitigation: Conditional Resource Loading and Component Ordering

To address this frontend-backend interaction vulnerability, several changes were implemented on the frontend:

- **Conditional `/api/config` Fetching in Gallery Context:** The primary fix involved modifying `src/contexts/gallery.ts` to ensure that the configuration resource (which fetches from `/api/config`) is only created and fetched when the user's authentication state (`isLoggedIn`) is true. This was achieved by making the `createResource` call conditional on the `isLoggedIn` signal from `useAppContext`. This prevents the unauthenticated request from ever being made.

```
    const authFetch = useAuthFetch();
    const { isLoggedIn } = useAppContext();

    // Data sources and actions
    const [config, { refetch: refetchConfig }] = createResource(
      isLoggedIn,
      async (loggedIn) => {
        if (loggedIn) {
          const fetcher = authFetch;
          const response = await fetcher("/api/config");
          const configData = await response.json();
          console.debug("Config loaded:", configData);
          return configData as ConfigResponse;
        }
        return undefined; // Return undefined when not logged in
      },
      {
        initialValue: undefined,
      }
    );

```

Listing 8: Conditional Config Resource Creation in `src/contexts/gallery.ts`

- **Corrected Component Rendering Order:** During debugging, it was also identified that certain components, such as `ReynardAssistantWrapper` and `DebugOverlayWrapper`, were being rendered outside the scope of `GalleryProvider` in `src/main.tsx`, leading to "useGallery must be used within a GalleryProvider" errors. These components implicitly or explicitly rely on the `GalleryContext`. The `src/main.tsx` file was updated to ensure these components are nested within the `GalleryProvider`, guaranteeing that they have access to the necessary context.

```
    <AppProvider>
        <CaptionerProvider>
            <AuthGuardWrapper>
                <GalleryProvider>
```

```
                    <GlobalIndexingMonitor />
                    {props.children}
                    <ReynardAssistantWrapper />
                    <DebugOverlayWrapper isVisible={isOverlayVisible()} />
                </GalleryProvider>
            </AuthGuardWrapper>
            <NotificationContainer />
            <ConnectionMonitor />
        </CaptionerProvider>
    </AppProvider>

```

Listing 9: Updated Component Hierarchy in `src/main.tsx`

Analysis: The implementation of conditional resource loading and corrected component rendering order successfully resolved the `401 Unauthorized` issue during login. The frontend now gracefully handles authentication states, fetching protected resources only when a user is authenticated. This ensures a smooth and secure login experience, highlighting the importance of securing backend endpoints and respecting frontend logic for authentication and component dependencies.

9 Vulnerability Severity and Prioritization

To effectively allocate resources and address identified vulnerabilities, future assessments will rank findings based on a severity matrix that considers both the technical impact and business impact. We will adopt a common vulnerability scoring system (e.g., CVSS) where applicable, and assign a remediation timeline based on severity:

- **Critical:** Immediate remediation required (e.g., unauthenticated RCE, data breach).
- **High:** Remediation within 1-7 days (e.g., authenticated RCE, significant data exposure).
- **Medium:** Remediation within 1-4 weeks (e.g., privilege escalation, DoS vulnerability).
- **Low:** Remediation within 1-3 months (e.g., information disclosure, minor misconfigurations).
- **Informational:** No immediate remediation required, but recommendations for best practices.

9.1 Vulnerability: Inadequate Input Validation for User Registration (`/api/register`)

- **Endpoint:** `/api/register`

- **Method:** POST
- **Description:** The fuzzer revealed that the /api/register endpoint, responsible for new user registrations, exhibited inconsistent error handling and acceptance of various malformed or invalid inputs for username and password fields. While some fuzzed inputs correctly resulted in 422 Unprocessable Content due to Pydantic's min_length and max_length validations (e.g., empty strings, overly long strings, or very short strings), other inputs led to 400 Bad Request or unexpected behavior. Specifically, inputs like 00 (null byte) or certain special characters, when passed as part of the username or password, could potentially bypass intended string validation or lead to unhandled exceptions downstream, indicating a lack of comprehensive input sanitization beyond basic length checks. The fuzzer output showed a mix of 422 Unprocessable Content for validation errors and 400 Bad Request potentially for issues like existing usernames or other data source constraints.
- **Impact:** Inadequate input validation can lead to several security risks, including: storing malformed or malicious data in the database, potentially enabling injection attacks (e.g., SQL injection if inputs are not properly escaped in database queries), denial-of-service by causing backend errors with specially crafted inputs, and account enumeration if error messages distinguish between non-existent and existing usernames. The inconsistent error codes also make it harder for the frontend to gracefully handle registration failures.

9.2 Mitigation: Enhancing Input Validation and Error Handling for User Registration

To address the input validation issues in the /api/register endpoint, comprehensive validation will be implemented for both username and password fields, extending beyond simple length checks to include character set restrictions and more specific format validations where appropriate. Error responses will be standardized to provide clear and consistent feedback to the client. This will involve:

- **Username Validation:** Enforcing stricter regex patterns for usernames (e.g., allowing only alphanumeric characters, underscores, and hyphens) to prevent injection of special characters or control characters that could cause issues.
- **Password Complexity:** Implementing additional password complexity requirements (e.g., requiring a mix of uppercase, lowercase, numbers, and symbols) to enhance security.
- **Standardized Error Responses:** Ensuring that all validation failures return a consistent 422 Unprocessable Content with clear detail mes-

sages, and that application-level errors (like username already exists) return `400 Bad Request` with specific error messages, avoiding generic `500 Internal Server Error` where possible.

9.3 Vulnerability: Insufficient Input Validation for Engagement Record (`/api/engagement/record`)

- **Endpoint:** `/api/engagement/record`
- **Method:** POST
- **Description:** The fuzzer identified that the `/api/engagement/record` endpoint, which accepts user engagement data, is vulnerable to ‘422 Unprocessable Content‘ errors when receiving malformed inputs. This occurs because the ‘EngagementRecord‘ Pydantic model expects specific data types for its fields (‘username: str‘, ‘engagement$_t$*ype* : *str*‘, ‘*value* : *float*‘, ‘*metadata* : *Optional*[*Dict*[*str*, *Any*]]‘).*However, thefuzzersuccessfullysentnon*− *conformingdata, suchasstringsforthe*‘*value*‘*field*(*whichexpectsafloat*)*orlists/stringsforthe*‘*metadat* *sideprocessingbeforevalidationfails.*
- **Impact:** This vulnerability primarily affects the robustness and reliability of the engagement tracking system. Malformed requests can lead to ‘422 Unprocessable Content‘ errors, which, while preventing incorrect data from being stored, indicate that the API is not gracefully handling unexpected input formats at an earlier stage or providing more specific feedback. Persistent malformed requests could contribute to server load or obscure more critical issues in logs. While not a direct security exploit in itself (as sensitive data is not exposed and the server doesn’t crash), it highlights an area for hardening input validation to improve API stability and maintainability.

9.4 Mitigation: Enhancing Input Validation for Engagement Record

To mitigate the input validation issues in the `/api/engagement/record` endpoint, the ‘EngagementRecord‘ Pydantic model will be enhanced with stricter validation rules, including:

- **‘engagement**$_t$*ype*‘*Validation* : *If*‘*engagement*$_t$*ype*‘*isexpectedtobefromapredefinedsetofvalues, itshou*
- **‘username‘ Field: While currently a ‘str‘, adding ‘min**$_l$*ength*‘*and*‘*max*$_l$*ength*‘*constraintsoraregex*
- **Robust Error Handling: Ensuring that all validation errors consistently return ‘422 Unprocessable Content‘ with clear, detailed messages, guiding clients on how to correctly format their requests.**

9.5 Vulnerability: Unauthenticated Captioning Service Access

- **Endpoints:** `/api/joycaption/download-status` (GET), `/api/caption-types` (GET), `/api/caption-types` (POST), `/api/generate-caption/{path}` (POST), `/api/batch-transform-captions` (POST), `/api/batch-generate-captions` (POST)
- **Description:** Several endpoints related to the captioning service, including checking JoyCaption model download status and managing caption types, were found to be accessible to unauthenticated users. This posed a security risk as it could lead to information disclosure (e.g., model download progress) or unauthorized configuration changes (e.g., updating caption types) by guest users. Additionally, the frontend was proactively attempting to fetch/sync these details even when logged out, leading to undesirable console messages and unnecessary backend requests.
- **Impact:** This vulnerability could allow unauthenticated users to gather sensitive operational information about the backend's captioning capabilities and model download status. More critically, it could allow for unauthorized modification of caption type configurations, potentially disrupting legitimate application functionality or leading to data integrity issues. The continuous frontend calls also generated distracting console warnings for guest users.

9.6 Mitigation: Backend Authentication and Frontend Conditional Logic

To mitigate the unauthenticated access to the captioning service and associated frontend issues, the following measures were implemented:

- **Backend Endpoint Protection:** The `/api/joycaption/download-status` (GET), `/api/caption-types` (GET), and `/api/caption-types` (POST) endpoints in `app/main.py` were updated to require authentication via the `Depends(get_current_user)` dependency. This ensures that only authenticated users can access or modify these resources.
- **Frontend Conditional Fetching (JoyCaption Download Status):** The `checkJoyCaptionDownload` function and its associated `createEffect` polling mechanism in `src/contexts/app.tsx` were modified. The check for JoyCaption download status now includes a condition to only proceed if `store.isLoggedIn` is true. This prevents unnecessary API calls when the user is logged out.
- **Frontend Conditional Sync (Caption Types):** The `syncCaptionTypesToBackend` function in `src/contexts/captionTypes.tsx` was updated. It now checks the `isLoggedIn` state (passed as a boolean parameter) and will skip the synchronization process if the user is not logged in. This prevents the

frontend from attempting to update backend caption types without authentication.

Analysis: By implementing authentication on the backend endpoints and introducing conditional logic in the frontend for relevant data fetches and synchronizations, the captioning service is now protected from unauthenticated access. This not only enhances security by preventing unauthorized information disclosure and configuration changes but also improves the user experience by eliminating misleading console messages for guest users. The system now gracefully handles authentication states, ensuring that sensitive operations are only performed when appropriate user privileges are established.

10 Future Work and Threat Model Evolution

Security is an ongoing process, and the threat landscape continuously evolves. Future work will include a living-document approach to periodically reassess the threat model and incorporate new attack techniques and vulnerabilities as they emerge. This will involve:

- **Regular Threat Intelligence Review:** Staying updated with the latest attack vectors, exploits, and security trends relevant to web applications and Python-based backends.
- **Supply Chain Security:** Assessing the security of third-party dependencies and libraries to mitigate risks associated with vulnerable components.
- **Server-Side Template Injection (SSTI):** Investigating potential SSTI vulnerabilities, especially if templating engines are introduced or modified.
- **Regular Penetration Testing:** Conducting periodic manual and automated penetration tests to ensure continued security posture.
- **Integration with Security Tools:** Exploring integration with advanced security tools for static application security testing (SAST) and dynamic application security testing (DAST).

11 Recommendations and Mitigations

Based on the findings, this section will provide actionable recommendations for patching identified vulnerabilities and improving the overall security posture of the Reynard backend. For all implemented mitigations, it is crucial to add CI-gated security tests and ensure comprehensive test coverage for all protected routes to prevent regressions.

11.1 Mitigation: Restricting Access to `/api/config`

To address the "Unauthenticated Information Disclosure" vulnerability, the `/api/config` endpoint has been secured by enforcing administrator-level authentication. This was achieved by adding a dependency on the `is_admin` function to the endpoint's route decorator.

```
@app.get("/api/config")
async def get_config(current_user: User = Depends(is_admin)):
    response = {
        # Basic config from file
        **config.to_dict(),
        "security_flag": "FLAG{YOU_FOUND_THE_CONFIG_FLAG}", # Our secret flag
        # Runtime information
        "tags_file_path": CUSTOM_TAGS_FILE_PATH,
        "root_dir": str(ROOT_DIR),
    }
```

Listing 10: Security Fix in `app/main.py`

This change ensures that only users with the 'admin' role can successfully access the `security_flag` within the the `/api/config` response, thereby preventing unauthenticated information disclosure.

References

[1] *Pen Testing a City*, 2015.

[2] Esra Abdullatif Altulaihan, Abrar Alismail, and Mounir Frikha. A survey on web application penetration testing. *Electronics*, 12(5):1229, 2023.

[3] Chiem Trieu Phong and Wei Qi Yan. An overview of penetration testing. *International Journal of Digital Crime and Forensics (IJDCF)*, 6(4):50–74, 2014.

[4] Chan Wai. Conducting a penetration test on an organization. *SANS Institute Whitepaper*, 2001.

12 Presentation Polishing

To enhance the readability and impact of this report, future revisions will incorporate:

- **Updated Bibliography:** Including more recent references on automated penetration testing frameworks and security in Python-based microservices.
- **Visual Aids:** Integrating figures (e.g., architecture diagrams, attack flows) and tables (e.g., summarizing endpoints tested vs. outcomes, vulnerability matrices) to improve clarity and provide quick insights into complex information.

13 New Automated Audit Scripts

To complement prior tests, we added three non-destructive audit scripts under `blackhat/` and executed them against the running backend.

13.1 Upload Path Traversal Probe

Script: `path_traversal_upload_test.py`

This script authenticates and attempts to upload a valid image with a crafted filename containing parent directory segments (e.g., `"../code/yipyap/blackhat/traversal_poc.png"`). It then verifies whether a file was written outside `ROOT_DIR`.

Result: No file was created outside `ROOT_DIR`. The server constructs the destination using a resolved target directory and does not interpret path separators embedded in the uploaded filename as a directory path. This suggests traversal via filename is mitigated for the upload path.

Recommendation: Optionally normalize per-file `filename` by applying a basename and sanitization (e.g., `Path(file.filename).name`, combined with a whitelist) before write to further harden against path separator edge-cases in certain server/proxy stacks.

13.2 Arbitrary Output Path Write Probe

Script: `arbitrary_write_output_path_test.py`

This script authenticates and posts to endpoints that accept an `output_path` (audio/video/text operations) using absolute paths outside `ROOT_DIR` under the repository tree, then checks if files were created.

Result: No files were created at the absolute paths during our test runs. Endpoints generally fail earlier due to nonexistent source inputs, and they do not proceed to write outputs. However, several endpoints take raw paths and only check existence of sources using `Path(...)` rather than `resolve_path(...)`, which indicates a potential trust boundary mismatch if future flows create outputs unconditionally.

Recommendation:

- Enforce path resolution with `resolve_path` for both source and output paths across `/api/audio`, `/api/video`, and `/api/text` modules.
- Gate writes by validating intended destinations are *within* `ROOT_DIR` before any `mkdir` or file creation.

13.3 CORS Configuration Probe

Script: `cors_check.py`

Sends an OPTIONS preflight with a foreign `Origin` and inspects *Access-Control-Allow-** headers.

Observation: The backend currently configures CORS with `allow_origins=["*"]`, `allow_methods=["*"]`, `allow_headers=["*"]`. While convenient for development, this is overly permissive for production. If credentials are ever enabled, combining `*` with credentials would be unsafe.

Recommendation (Production):

- Restrict `allow_origins` to the exact frontend domain(s).
- Maintain credentials policy consistent with token storage (avoid cookies across origins unless strictly necessary with explicit origins).

13.4 Summary

These scripted probes confirm that filename-based traversal in uploads appears mitigated, detected no out-of-root writes via `output_path` in current flows, and highlight the need to lock down CORS in production. We recommend standardizing path validation with `resolve_path` across all endpoints that read or write files and tightening CORS configuration for deployment.