DANZO Audit Report

Friday, 9 May 2025

1. Executive Summary

- **Purpose of the Audit**
  This audit provides a comprehensive examination of the closed-source backend code for the decentralised application running on the Cardano blockchain under the domain danzo.gg. The primary goal is to identify any potential loopholes or vulnerabilities that malicious actors could exploit to extract liquidity or otherwise compromise the system.

- **Overall Assessment**
  Overall, the backend exhibits a sound structural design. However, we did identify several security and reliability concerns that require further attention.

- **Key Findings**

  - **No initial withdrawal limits:** Initially, the source code did not enforce any upper withdrawal thresholds, potentially allowing the entire liquidity to be withdrawn.
  - **Audit-driven improvements:** As a direct result of the audit, withdrawal limits and multi-stage validation checks were introduced.
  - **Triple-verification process:** Currently, every withdrawal undergoes a triple-verification procedure before final execution, significantly reducing the risk of large-scale unauthorized withdrawals.

2. Scope of Work

- **Code and Components Audited**
  We examined the entire `casino` directory within the closed-source DANZO GitHub repository, focusing on its deposit, withdrawal, profit management, and concurrency check functionality.

- **Exclusions**
  Our review was limited to the custom backend transaction-handling logic. Integrated libraries—such as native Python modules, PyCardano, and certain Emurgo support libraries—were outside the scope of this audit.

- **Time Frame**
  The audit commenced in January 2025 and was completed by May 2025.

3. Methodology

- **Review Process**
  - **Manual Code Review and Developer Interviews:** We conducted an extensive manual review of the codebase, supplemented by interviews with developers to clarify areas involving complex Cardano blockchain logic.
  - **Environment and Testing:** Transaction testing was primarily performed on a dedicated testnet environment. Concurrently, we monitored live logs and behavior on the mainnet, as the application continued operating throughout the audit.

- **Risk Rating Methodology**
  - In this audit, we used two risk categories:
  - **High:** Represents serious issues that, if exploited, could lead to significant financial, security, or operational harm. These require immediate attention and remediation.

- **Not Important:** Designates minor or negligible concerns that pose little to no immediate risk. They may be addressed at the discretion of the development team.
-

4. Detailed Findings

4.1 Architecture Overview

The backend is primarily organized around a **continuous transaction-handling loop** that monitors incoming operations (e.g., deposits, withdrawals) and processes them in real time. Below are the major logical components:

1. **Main Transaction Handler (Callback)**
   - Reads newly processed transaction references and evaluates whether the operation is a deposit, withdrawal, or liquidity update.
   - Invokes utility functions for validation (e.g., `get_balance`, `getWithdrawableBalance`) and constructs Cardano transactions via **PyCardano** or updates the balances in the Database to either credit or debit player balances.

2. **Liquidity & Profit Management**
   - Deposit/Withdrawal Routines:
     - `Proccess_ADA_Deposit` manages liquidity deposits in ADA, tracking them in a secure Database environment.
     - `withdraw_casino_liquidity` handles logic to remove contributed liquidity, applying penalty checks if the user withdraws too soon.
   - Casino Profit Tracking:
     - The system calculates total liquidity and potential profits at set intervals (e.g., every poll). Profits for specific tokens (e.g., DANZO) are recorded in secure JSON files, with concurrency controls in place to prevent race conditions during read/write operations.

3. **Reward Distribution**
   - On a timed schedule (checked every few polls), if a threshold (like five days) has passed, the `Distribute_rewards` function calculates any leftover profits to be distributed to liquidity providers.
   - The system then calls `Map_profits_to_participants` to determine proportional payouts, builds a transaction, and appends the distribution data to a separate engine (ORCA LABS) for final disbursement.

4. **Concurrency & File-Based State**
   - Certain operations (e.g., updating JSON file, liquidity points, or cooldown limits) use **fcntl**-based file locking to avoid race conditions.
   - MongoDB is used in tandem to store player balances and transaction logs, with JSON files serving as supplemental or historical trackers.

5. **Supporting Modules and Checks**
   - **Cooldown & High-Value Logic:** Functions like `can_withdraw` and `is_token_high_value` apply thresholds (e.g., X-hour windows, Yk ADA limit) to deter large-scale or rapid withdrawals. To safeguard sensitive security information, details regarding X and Y remain undisclosed.

- **Hall of Fame & Stats:** Periodic scripts (e.g., `generate_hall_of_fame_announcements`) tally player statistics and record top performers.
- **Social Media & Email Notifications:** The backend integrates with external services (Discord, Twitter, Mailjet) to broadcast significant events or alerts (e.g., large deposits, blocked withdrawals).

Overall, the backend's structure centers on **transaction event processing**, **liquidity management**, **reward scheduling**, and **real-time validations**, with all modules coordinating to maintain a decentralized casino environment on Cardano.

External Services and Dependencies

- **BlockFrost API**
  The application leverages the BlockFrost service to query on-chain data (e.g., balances, transaction details) and to submit transactions to the Cardano mainnet. Primary and fallback project IDs are utilized, ensuring uninterrupted service even if one key reaches its rate limit.

- **PyCardano**
  A Python library used to build, sign, and submit Cardano transactions programmatically. It handles low-level transaction assembly, key generation, and other blockchain interactions.

- **TapTools API**
  External API calls retrieve live token price data in ADA/USD to assess the value of deposits, withdrawals, and large transaction attempts. The application checks and integrates these prices in real time for risk-based decisions (e.g., blocking large withdrawals).

- **Mailjet / SMTP**
  The code integrates with Mailjet's SMTP services to send automated notifications and alerts (e.g., high-value withdrawals or admin alerts). Credentials are loaded from a secure storage mechanism and environment variables.

- **MongoDB**
  A local MongoDB instance is used to store essential data, such as player balances and transaction histories. The application references a connection URI (redacted in production) and issues queries/updates via the `pymongo` client library.

- **Tweepy (Twitter)**
  Twitter updates about significant deposits or milestone achievements are posted using Tweepy. OAuth credentials are correctly managed outside the code, in environment variables, for production to prevent accidental key exposure.

- **Discord Posting**
  A custom HTTP POST endpoint ties into a separate Flask service that posts updates, logs, or images to Discord channels. Proper error handling (timeouts, retry logic) ensures robust communication with Discord.

Handling & Security

- **Credential Management**
  Keys and passwords used in the snippet are stored securely (e.g., environment variables, encrypted vaults) to prevent unauthorized access. The system operates within an isolated virtual machine environment, and SSH access is strictly controlled, allowing remote connections exclusively for authorized administrators.

- **HTTPS and Error Handling**
  All external calls (BlockFrost, TapTools, Mailjet, etc.) are made over HTTPS with appropriate exception handling to manage potential downtime or transient errors safely.
- **File Locking and Concurrency**
  Operations that update local JSON files (e.g., deposit logs or cooldown trackers) employ file locking (`fcntl`) to avoid race conditions when multiple processes attempt to read/write simultaneously.

Overall, the application demonstrates mindful integration with external services. Actual credential values and internal configurations are not publicly shared, following best practices to avoid exposing sensitive information in the codebase or audit report.

4.2 Code Quality and Maintainability

- Coding Standards
  The project exhibits a generally consistent style in terms of function and variable naming. That said, certain sections could benefit from additional refactoring to ensure uniform code patterns across all modules, particularly regarding function length and grouping logically related constants in a dedicated config file or class. Because this code is proprietary and not intended for open-source release, addressing the noted comment is considered a lower priority and does not pose a significant concern at this time.

- **Documentation and Comments**
  The code includes straightforward, inline comments for most functions, explaining parameters and return types. However, complex or multi-step logic (e.g., distribution engine appends, liquidity calculations) would benefit from more explicit high-level comments or brief design notes. Strengthening these explanations, especially around the concurrency handling or data validation steps, would improve long-term maintainability and onboarding for new developers.

- **Error Handling and Logging**
  Errors are handled fairly well through try-except blocks and custom exceptions. For instance, calls to external APIs (BlockFrost, Discord, TapTools) include retries or fallback logic, reducing the likelihood of crash loops. However, logs are often printed to stdout (e.g., `print("Error...")`). Integrating a more centralised logging framework (with levels like DEBUG/INFO/WARN/ERROR) would offer improved tracking, rotation, and alerting when running in production.

- **Modularity**
  Key functionality (e.g., transaction building, withdrawal checks, external notifications) is somewhat separated, but there is still room for further decoupling. For example:
  - **Transaction Logic**: The callback function (`callback`) handles both deposit/withdraw logic and direct calls to external APIs (e.g., emailing, Discord posting). Splitting these responsibilities into distinct modules or services (e.g., a “notification service” vs. a “transaction service”) would foster clearer boundaries and testability.
  - **Email & Discord Notifications**: There is a basic separation through dedicated functions (`notify_admin`, `post_to_discord`), yet integrating them into a more generic “Alert” interface would make future expansions (e.g., additional channels) simpler.

Overall, the code is coherent and workable, but future maintainability would benefit from cleaner modular boundaries, more uniform documentation, and a centralised logging approach.

4.3 Security Controls

Transaction Validations

- **ADA vs. Custom Tokens**
  The code differentiates between **ADA (lovelace)** and **custom tokens** during deposits and withdrawals. ADA liquidity deposits invoke `Proccess_ADA_Deposit` with a specified minimum (`min_ada_deposit`) and maximum (`max_ada_deposit`), ensuring no one can deposit less than the required threshold or exceed the allowed max.
  - **Prior State:** Initially, there were no strict upper withdrawal limits, allowing an entire liquidity pool to be emptied in a single transaction.
  - **Post-Audit Implementation:** This audit introduced upper limits for withdrawals and a multi-step verification (e.g., triple-check logic) to safeguard large transaction outflows.

Double Checks and Limits

- **Cooldown and Thresholds**
  - `COOLDOWN_DURATION` (X hours) and `ADA_THRESHOLD` (Yk ADA equivalent) are enforced to block large cumulative withdrawals in a short time window.
  - **Effectiveness:** These checks significantly mitigate rapid "drain attacks." If a user's cumulative withdrawal within X hours exceeds the threshold, the code prevents execution and sends admin alerts.
  - **Audit Impact:** These controls did not exist initially and were added following identified vulnerabilities.

Concurrency and File Locking

- **File Locking (`fcntl.flock`)**
  - The application updates JSON state files (e.g., `withdrawal_cooldowns.json`) with **exclusive file locks** to prevent race conditions from concurrent read/write operations.
  - **Recommendation:** Although effective for lower concurrency, scaling up may benefit from fully **database-based** transactions and locks (e.g., MongoDB transactions) to reduce complexity and eliminate potential file-locking bottlenecks.

Sensitive Data Management

- **.env Usage**
  - Credentials for SMTP (Mailjet), BlockFrost, and other APIs are read from environment variables.

Database Security

- **MongoDB Connection**
  - The code connects to MongoDB via a URI including username and password. Basic authentication is present.
  - **Best Practices:** Ideally, SSL/TLS should secure connections, and network access to the database should be firewalled. Role-based access control and regular credential rotation are also recommended for production deployments.

External API Error Handling

- **Retries and Exceptions**

- External requests (BlockFrost, TapTools, Discord, Twitter) are wrapped in try-except blocks. Certain calls include configurable retry loops or fallback keys (e.g., primary vs. backup BlockFrost project IDs).
- **Recommendation:** Introduce a unified retry/backoff mechanism for all external services to ensure consistent error handling. Logging these errors in a centralised log or monitoring tool would provide better visibility into repeated failures.

4.4 Potential Vulnerabilities

Withdrawal Bypass

- X-Hour and Y% Threshold
  - The main concern is whether attackers can split a large withdrawal into many small ones to circumvent the X-hour or Y% checks.
  - **Current Safeguards:** The code tracks cumulative withdrawals (`can_withdraw`) and denies any attempt that pushes the sum over the threshold within the cooldown window.
  - **Severity:** Considered **High** if the code is not tightly enforced. Currently, no direct bypass was found, but thorough testing is recommended to confirm no edge cases exist.

Race Conditions

- Simultaneous Deposits/Withdrawals
  - While `fcntl.flock` mitigates file-level race conditions, highly concurrent environments or future scaling could expose timing issues, especially if large volumes of transactions occur simultaneously.
  - **Risk Level: Medium to High** for scalability. In the present design, careful locking reduces immediate risk, but remains a key focus for future audits.

Replay Attacks

- Processed Transactions Log
  - The code maintains a file to detect and ignore previously processed transaction hashes.
  - **Recommendation:** Periodically rotate or prune this file, and ensure consistent checks for duplicates, especially if `tx_hash` lifetimes overlap with reorgs on the Cardano chain.

Insufficient Validation

- Addresses and Amounts
  - Some basic validation exists (e.g., checking `check_unit_exists` for tokens, verifying deposit/withdraw amounts). However, deeper checks (e.g., verifying address format or ensuring user is authorized for founder-only withdrawals) rely on external calls and API requests (e.g. using block frost API to fetch stake addresses).
  - **Recommendation:** Additional strict validation on user input or suspicious addresses to prevent injection or manipulation. It is important to note that, prior to the audit, the system was vulnerable to Franken-address attacks. However, this issue has since been successfully patched.

4.5 Remediations and Recommendations

Immediate Changes Implemented

- **Cooldown Logic & Withdrawal Limits:** Introduced withdrawal cooldowns (X-hour window) and a threshold to prevent high-value tokens or ADA from being drained.
- **Concurrency Locking:** Ensured file locks around JSON writes, reducing data corruption risk.
- **Enhanced Notifications:** Implemented administrative email and Discord alerts when thresholds are exceeded.

Short-Term Recommendations

1. **Refine Logging:** Introduce a unified logging system (e.g., Python `logging` with DEBUG, INFO, WARN, ERROR) to replace scattered `print` statements.
2. **Further Input Validation:** Ensure all user-facing routes and internal calls validate amounts and addresses more rigorously.
3. **Minor Refactoring:** Group constants (keys, threshold values) in a single config file or environment-based dictionary for easier updates and maintenance.

Long-Term Recommendations

1. **Database-Centric Concurrency:** Migrate file-based data tracking to MongoDB transactions or another robust DB solution for more scalable concurrency control.
2. **Formal Testing & CI/CD:** Increase code coverage with automated tests (unit/integration), and enforce CI/CD checks for each PR to detect regressions before deployment.
3. **Extended Security Program:** Consider launching a bug bounty or security disclosure program. This encourages external security researchers to responsibly report vulnerabilities, improving overall reliability in a financial-grade environment.
    -

5. Conclusion

Overall Security Posture

With the newly introduced withdrawal limits, cooldown mechanisms, and improved validation checks, the backend now offers significantly stronger defenses against large-scale unauthorized withdrawals. Critical liquidity management functionalities have been fortified to prevent a single transaction or rapid series of transactions from draining the system. Furthermore, concurrency locking, more robust error handling for external APIs, and improved notification channels collectively enhance reliability and transparency.

Key Takeaways

- Critical Fixes Implemented:
    - **Withdrawal Thresholds & Cooldowns:** These changes now prevent attackers from exploiting large or repeated withdrawals.
    - **File Locking & Validation Checks:** Race conditions during deposit/withdraw scenarios are mitigated by exclusive locks, while validation steps block suspicious activity.
- Strengths Identified:
    - **Consistent Code Style:** Despite a few minor deviations, the overall structure and naming conventions remain largely uniform.
    - **Up-to-Date Dependencies:** Core libraries (e.g., PyCardano, pymongo, and external API integrations) are current, minimizing vulnerability exposure from older versions.

Next Steps

1. **Periodic Security Audits:** We recommend scheduling a follow-up audit within **6–12 months** or whenever major architectural changes are introduced.
2. **Scalability Planning:** As user volume grows, consider a database-centric approach to concurrency to reduce reliance on file-level locking.
3. **Continued Monitoring & Updates:** Regularly review API keys, dependencies, and logs to stay ahead of potential threats and maintain a secure operating environment.

6. Disclaimer

The application running on the domain **danzo.gg** comprises multiple components, many of which fell outside the scope of this audit. In particular, the entire game logic was excluded, as it emulates standard casino games using real-world probabilities—an area the auditor could neither fully evaluate nor guarantee. However, to the best of our knowledge, the application utilizes reputable libraries to ensure randomization, without incurring additional liability for their correctness. This audit focused exclusively on financial handling and critical security aspects. The game logic is assumed correct as a prerequisite, and we accept no responsibility for the accuracy of any game payouts; as with traditional casinos, payouts may be voided in the event of a malfunction. Given the high-risk nature of this application, all interactions should be undertaken only after careful consideration of potential risks. Any modifications made to the audited backend repository after **May 8th** were not assessed as part of this audit.