sig9.ch
Dienerstrasse 15
8004, Zürich, Switzerland
info@sig9.ch

Audit Report

TO:
KongSwap

AUDIT ID
643ddf15-90ef-4183-859a-832c86225095
AUDIT VERSION
2.0 (review)
LAST UPDATE
07/10/24
[43e82e9ded8cbbbcac6422c0b209cccab1230fca]

Project description and audit scope

Auditor: Michele Federici
Audit type: White Box
Report type: Public
Language: Rust
Ecosystem: ICP
Github: github.com/KongSwap

KongSwap is a decentralized exchange based on the Internet Computer's network and technologies.

This report provides an overview of the security analysis performed on KongSwap's codebase, a follow-up review of implemented fixes and recommendations, as well as the targets and methodologies of the tests performed.

The backbone of the DApp, **kong_backend**, is an ICP canister written in the Rust language and implementing all the backend logic, exposing the public API, and managing the platform's storage.

The goal of this audit was to review the design and logic, the robustness of safety controls, and to identify conditions potentially leading to exploitable vulnerabilities, undesired behaviors, and service disruptions.

The analysis was performed over the entire project, in a white box fashion and without scope boundaries, thus having full access to the code, documentation, and internal knowledge.

The auditing process consisted of both static code review and dynamic penetration testing. All the web-facing endpoints have been tested for insufficient input validation, inadequate permission controls, and other flaws that could lead to malicious exploitation of the platform, accounts, and funds.

Please note that neither this audit, nor any other, will ever guarantee that a system is immune to all security threats. Cybersecurity is a boundless and imperfect field that requires consistent efforts; at any time, unseen vulnerabilities may be detected and new ones may be introduced. Nevertheless, periodic external audits provide important outside perspectives, insights, and a detailed snapshot of the current security posture of a system.

Findings summary

The subsequent sections of this report detail the findings, their impact, and the recommended or implemented remediations. The findings are classified on the following severity classification system: Info, Low, Medium, High, Critical.

Info: 4

Low: 2

Medium: 2

High: 0

Critical: 0

Re-entrancy, double-spend, and state pollution analysis

No vulnerabilities identified

Safety controls

Adequate emergency controls in place

Project dependencies

No known vulnerabilities found

Findings details

[M1] Heavy and unbounded use of stable memory

- **Severity:** Medium
- **Status:** Partially addressed

Description

The application utilizes stable memory extensively without implementing limits or cleanup mechanisms. Data structures are allowed to grow indefinitely, which can, over time, lead to resource exhaustion and potentially open a window for DOS attacks.

Impact

- **Service disruption:** memory exhaustion may cause the application to crash or become unresponsive.
- **Denial of Service (DoS):** adversaries could exploit this by creating excessive data to consume resources.
- **Performance degradation:** increased memory usage can slow down processing and response times.

Recommendation

Introduce, when possible, memory management strategies such as:

- **Cleanup mechanisms:** regularly remove old and unnecessary data.
- **Data capping:** set limits on the number of objects stored (e.g., implement FIFO).
- **Data clustering:** organize data into smaller clusters to optimize access and reduce memory overhead.
- **Archiving:** move obsolete data to different backup structures, less affected by this problem.

[M2] Use of token symbols instead of their addresses

- **Severity:** Medium
- **Status:** Fixed

Description

Token symbols should only be used as human-friendly labels. The use of symbols for identification, filtering, or selection, should generally be avoided in favor of their addresses, which are guaranteed to be unique.

Impact

- **Transaction errors:** misidentification of tokens could result in incorrect transactions.
- **Security risks:** Potential exploitation by malicious actors to deceive users or the system.

Recommendation

Ensure that, whenever possible, selections and filtering operations use unique identifiers to prevent ambiguity and risks.

[L1] set_kong_settings function prone to human errors

- **Severity:** Low
- **Status:** Unfixed

Description

The set_kong_settings function uses an ambiguous JSON merging implementation with at least a couple of non obvious edge cases, like a recursive call on mutable arguments and the replacement (instead of merging) of the maintainers' array (kingkong).

Impact

- **Configuration errors:** settings may be incorrectly applied or lost.
- **Operational risks:** potential unintended misconfigurations.

Recommendation

Extract at least emergency fields into dedicated functions. For instance, it's always good to have a dedicated and simple call to toggle maintenance mode (safety switch).

[L2] Risk of inconsistent states and lock conditions caused by unhandled exceptions

- **Severity:** Low
- **Status:** Fixed

Description

Unhandled exceptions within update calls could cause the system to enter inconsistent states or lock situations..

Impact

- **Data inconsistency:** risk of inconsistent or incomplete data states.
- **Non trivial recovery:** it might be difficult to understand and recover certain cases.

Recommendation

Implement proper exception handling, especially in functions used in calls modifying the state.

[I1] Functions borrowing memory references being logically separated even when mostly used in conjunction, may increase the risk of introducing reentrancy vulnerabilities

- **Severity:** Informational
- **Status:** Unfixed

Description

Immutable and mutable memory references are consistently borrowed using specific functions (get, insert, update), even if often used sequentially. This separation can increase the risk for developers to inadvertently introduce operations between them, which could potentially lead to re-entrancy and double-spend scenarios.

Impact

- **Security risks:** accidental introduction of re-entrancy bugs in the gap between get and update operations.

- **Code maintainability:** increased stress to ensure that no intermediate operations are introduced.

Recommendation

Consolidate operations requiring both read and write access into single functions, abstracting away the problem and reducing the stress of being careful at every check & update operation.

[I2] Use guards for validation and access control

- Severity: Informational
- Status: Fixed

Description

Validation and access controls were performed using in-method functions instead of using guards, as per best practice. Other than code aesthetics, guards also have technical advantages, are optimized for their scope and less computationally expensive.

Impact

- Optimization: in-method functions are heavier, guards are executed before instantiating the actual function they are guarding and its content.
- Cleanness: the guard approach is clear and consistent.

Recommendation

Refactor validations and access controls functions to guards.

[I3] Expired or invalid user sessions causing infinite front-end loop

- Severity: Informational
- Status: Fixed

Description

Expired or invalid user sessions were causing the front-end to enter an infinite reload loop.

Impact

- User experience: users could not easily realize what was the problem and how to resolve it.

Recommendation

Implement proper session clearing mechanisms when a session is, or becomes, invalid.

[I4] Prevent cascading failures in utility scripts (bash)

- Severity: Informational
- Status: Unfixed

Description

The helpers scripts provided in the project continue executing subsequent commands when a prior command fails.

Impact

- Integrity: risk of unintended operations and unpredictable results and states.

Recommendation

A simple solution that doesn't require individual logic for each command is to just add `set -e` at the beginning of the scripts. This setting instructs bash to stop the execution whenever a command exits with a non-zero status, preventing further commands to run under faulty conditions.

Additional analysis

Re-entrancy, double-spend, and state pollution

An in-depth examination was conducted to identify potential vulnerabilities related to re-entrancy attacks, double-spending scenarios, and state pollution issues.

The analysis focused on how state mutations and intermediate states are managed during canister calls and internal operations.

Result

The project demonstrates effective handling of state mutations, ensuring that all state changes are atomic and occur in a controlled manner. When atomicity of an entire operation is not possible (e.g. inter-canister calls), intermediate states are properly set to prevent abuses.

Safety controls

The project implements sufficient emergency controls for rapid responses in critical situations. These controls empower the team to react swiftly to unforeseen issues, minimizing potential damage.

- **Maintenance mode (safety switch):** a global safety switch allows controllers to activate maintenance mode. When enabled, operations are halted.
- **Admin functions:** a set of admin functions is available, enabling controllers and maintainers (kingkong list) to perform actions such as pausing the platform, adjusting configurations, managing user accounts, performing emergency withdrawals, and other management tasks.

Project dependencies

An assessment of the external dependencies and the versions in use was conducted to identify any publicly known security vulnerabilities.

Result

- **Up to date:** all dependencies in use are updated to recent releases.

- **No known vulnerabilities**: at the time of the audit, none of the utilized versions in use had publicly disclosed vulnerabilities.