ANASTASIA LABS

Final Security Audit Report

Aquarium Protocol

Date December, 2024
Project ft-cardano-aquarium-sc
Version 1.0

Contents

Disclosure

This document contains proprietary information belonging to Anastasia Labs. Duplication, redistribution, or use, in whole or in part, in any form, requires explicit consent from Anastasia Labs.

Nonetheless, both the customer **Fluid Tokens** and Anastasia Labs are authorized to share this document with the public to demonstrate security compliance and transparency regarding the outcomes of the Protocol.

Disclaimer and Scope

A code review represents a snapshot in time, and the findings and recommendations presented in this report reflect the information gathered during the assessment period. It is important to note that any modifications made outside of this timeframe will not be captured in this report.

While diligent efforts have been made to uncover potential vulnerabilities, it is essential to recognize that this assessment may not uncover all potential security issues in the protocol.

It is imperative to understand that the findings and recommendations provided in this audit report should not be construed as investment advice.

Furthermore, it is strongly recommended that projects consider undergoing multiple independent audits and/or participating in bug bounty programs to increase their protocol security.

Please be aware that the scope of this security audit does not extend to the compiler layer, such as the UPLC code generated by the compiler or any areas beyond the audited code.

The scope of the audit did not include additional creation of unit testing or property-based testing of the contracts.

Assessment overview

From **Nov 9th**, **2024** to **Dec 5th**, **2024**, **Fluid Tokens** engaged Anastasia Labs to evaluate and conduct a security assessment of its **Aquarium** protocol. All code revision was performed following industry best practices.

Phases of code auditing activities include the following:

- **Planning** – Customer goals are gathered.
- **Discovery** – Perform code review to identify potential vulnerabilities, weak areas, and exploits.
- **Attack** – Confirm potential vulnerabilities through testing and perform additional discovery upon new access.
- **Reporting** – Document all found vulnerabilities.

The engineering team has also conducted a comprehensive review of protocol optimization strategies.

Each issue was logged and labeled with its corresponding severity level, making it easier for our audit team to manage and tackle each vulnerability.

Assessment components

Manual revision

Our manual code auditing is focused on a wide range of attack vectors, including but not limited to:

- UTXO Value Size Spam (Token Dust Attack)
- Large Datum or Unbounded Protocol Datum
- EUTXO Concurrency DoS
- Unauthorized Data modification
- Multisig PK Attack
- Infinite Mint
- Incorrect Parameterized Scripts
- Other Redeemer
- Other Token Name
- Arbitrary UTXO Datum
- Unbounded protocol value
- Foreign UTXO tokens
- Double or Multiple satisfaction
- Locked Ada
- Locked non Ada values
- Missing UTXO authentication
- UTXO contention

Executive summary

Aquarium is a decentralized protocol on the Cardano blockchain, designed to enable sponsored transactions, Babel fees, and schedulable transactions. Aquarium provides a solution for users and projects to offer liquidity through smart contracts, featuring an open network of Aquarium nodes that compete to execute scheduled transactions.

When a user creates a Rule (a condition that triggers a scheduled transaction). Aquarium nodes compete to execute the transaction at the appropriate time. Fees generated by the protocol are distributed among the registered nodes, which are required to stake a certain amount of `$FLDT` tokens to participate. This competitive model ensures fairness by redistributing revenues proportionally based on the actual work performed by each node.

The primary purpose of Aquarium is to facilitate sponsored and schedulable transactions within a secure and transparent environment. By integrating open-source smart contracts as a core component.

Code base

Repository

https://github.com/FluidTokens/ft-cardano-aquarium-sc

Commit

f805fb61f02f07ee86f08cb7ea84d06c38f2e054

Files Audited

SHA256 Checksum	Files
970c35d22057d225f9826111527ec237f afa0e4d932a888fbe9310c9492e0121	validators/parameters.ak
19638ebd5659839151ea5614367ea61a 97d219fd5010cef7f688fe32fc35eeab	validators/staker.ak
dc27e1dd224985acba67665c025ee4d cb25fa45912316506b73a0c9b51c087b5	validators/tank.ak

Severity Classification

- **Critical**: This vulnerability has the potential to result in significant financial losses to the protocol. They often enable attackers to directly steal assets from contracts or users, or permanently lock funds within the contract.
- **Major**: Can lead to damage to the user or protocol, although the impact may be restricted to specific functionalities or temporal control. Attackers exploiting major vulnerabilities may cause harmor disrupt certain aspects of the protocol.
- **Medium**: May not directly result in financial losses, but they can temporarily impair the protocol's functionality. Examples include susceptibility to front-running attacks,which can undermine the integrity of transactions.
- **Minor**: Minor vulnerabilities do not typically result in financial losses or significant harm to users or the protocol. The attack vector may be inconsequential or the attacker's incentive to exploit it may be minimal.
- **Informational**: These findings do not pose immediate financial risks. These may include protocol optimizations, code style recommendations, alignment with naming conventions, overall contract design suggestions, and documentation discrepancies between the code and protocol specifications.

Finding severity ratings

The following table defines levels of severity and score range that are used throughout the document to assess vulnerability and risk impact

Level	Severity	Findings
5	Critical	0
4	Major	0
3	Medium	0
2	Minor	3
1	Informational	4

Findings

ID-201 Missing Check for `min_to_stake` in `parameters.ak`

	Level	Severity	Findings
	2	Minor	Resolved

Description

In `parameters.ak`, there are no checks for `min_to_stake`. Even if the owner is performing the action, this value should not be negative.

Allowing `min_to_stake` to be negative could lead to unintended behavior or potential vulnerabilities.

Recommendation

Add a check when spending the parameter input to ensure that `datum_params.min_to_stake` is greater than zero. This validation prevents negative values and enhances the security of the contract.

```
// Missing check when spending the parameter input
// Expected condition:
datum_params.min_to_stake > 0
```

Resolution

Resolved in commit `46bf58f9610e8147c81590c527d5a4b878cf26ff`

ID-202: Hardcoded Reference Input Index in `staker.ak` Mint Function

	Level	Severity	Findings
	2	Minor	Resolved

Description

In `staker.ak` mint function, there is a hardcoded index used to access a reference input:

```
1  expect Some(referenceInput) = list.at(tx.reference_inputs, 0)
```
Aiken

Using a hardcoded value `(0)` is problematic if you have more than one reference input because inputs are ordered lexicographically by the ledger. This could result in the wrong reference input being used, introducing potential vulnerabilities.

Recommendation

Pass the index through the redeemer as input_indexes to make the implementation more flexible and eliminate conflicts arising from the lexicographical ordering of inputs based on ledger rules.

Resolution

Resolved in commit `46bf58f9610e8147c81590c527d5a4b878cf26ff`

ID-203: Hardcoded Reference Inputs in `tank.ak`

Level	Severity	Findings
2	Minor	Resolved

Description

In `tank.ak`, hardcoded indices are used to access reference inputs:

```
1 expect Some(referenceInput) = list.at(self.reference_inputs, 0)
2 expect Some(referenceInputParams) = list.at(self.reference_inputs, 1)
```

Aiken

Using hardcoded indices is problematic because reference inputs are ordered lexicographically by ledger rules. If the input order changes or additional inputs are added, it can result in incorrect references being accessed, potentially introducing vulnerabilities and disrupting contract functionality.

Recommendation

Allow the indices to be passed through the redeemer as input_indexes. This approach aligns with ledger ordering rules, ensuring flexibility and avoiding conflicts caused by hardcoded indices.

Resolution

Resolved in commit `46bf58f9610e8147c81590c527d5a4b878cf26ff`

ID-101: Redundant Computation in `staker.ak`

Level	Severity	Findings
1	Informational	Resolved

Description

In `staker.ak`, the following function is redundantly executed for every spend validator input:

```
1 let inputs_from_contract = count_inputs(self.inputs, input)
```
Aiken

Performing this computation repeatedly can lead to unnecessary computational overhead.

Recommendation

Delegate this computation to the minting policy instead of performing it for every spend validator input to reduce redundancy and improve efficiency.

Resolution

Resolved in commit `46bf58f9610e8147c81590c527d5a4b878cf26ff`

ID-102: Hardcoded Output Index in `staker.ak` Mint Policy

	Level	Severity	Findings
	1	Informational	Resolved

Description

In `staker.ak` mint function, there is a hardcoded index used to access transaction outputs:

```
1  expect Some(output_staking) = list.at(tx.outputs, 0)                    Aiken
```

Although outputs are not reordered by the ledger, relying on hardcoded indices can reduce interoperability with other protocols. This approach may cause issues if the output order changes, even though it does not pose a direct security risk.

Recommendation

Pass the index through the redeemer as output_indexes to enhance flexibility and improve interoperability with other smart contracts.

Resolution

Resolved in commit `46bf58f9610e8147c81590c527d5a4b878cf26ff`

ID-103: Code Optimization in `tank.ak`

Level	Severity	Findings
1	Informational	Resolved

Description

A code optimization is possible in **`tank.ak`** to simplify the validation logic. The current code:

```
1 // Current code
2 expect Some(stake_cred_signer) = batcher.stake_credential
3 expect Inline(stake_key_hash) = stake_cred_signer
4 expect VerificationKey(hash) = stake_key_hash
```

Aiken

Recommendation

Can be optimized to:

```
1 // Optimized code
2 expect Some(Inline(VerificationKey(hash))) = batcher.stake_credential
```

Aiken

Resolution

Resolved in commit `46bf58f9610e8147c81590c527d5a4b878cf26ff`

ID-104: Parameterization of Hardcoded Value in `tank.ak`

	Level	Severity	Findings
	1	Informational	Acknowledge

Description

The Cardano ledger enforces constraints to prevent excessive UTXO growth by requiring every UTXO entry to contain a minimum ADA value.

This minimum value is calculated using a formula based on the ledger protocol parameter `coinsPerUTxOByte`.

The issue arises when this minimum ADA value is hardcoded, as it prevents contracts from adapting to potential updates to the `coinsPerUTxOByte` parameter in future ledger upgrades.

Currently, a hardcoded value of `1500000` is used in `validate_tank_output` and `validate_payment_output`.

Recommendation

Parameterize the minimum ADA value within the contract parameters to future-proof the contract.

Document the calculation and purpose of this value for better clarity.

Resolution

Acknowledged