Réti Audit

Open Pooling

August 2024

By CoinFabrik

v202311

Executive Summary

CoinFabrik was asked to audit the contracts for the Réti Open Pooling project.

As well stated in its documentation[1], "the Réti Pooling protocol facilitates the creation of decentralized staking pools on the Algorand network, enabling groups of individuals to participate in consensus together. Users are able to trustlessly stake to a `validator` and earn yield based on the rewards the validator receives."

To achieve this, Réti developed two smart contracts, the `ValidatorRegistry` and the `StakingPool`. These smart contracts are part of a larger infrastructure that also includes the Réti Node Daemon[2] and a web app to facilitate operations for both stakers and validators.

During this audit we found one high issue, four medium issues and several minor issues. Also, an enhancement was proposed.

Seven issues were resolved, three were acknowledged and one was mitigated. The enhancement was implemented.

Scope

The audited files are from the git repository located at https://github.com/TxnLab/reti. The audit is based on the commit `1e2564b2df94657d86b7730d2b2d87f136b278a1`. The fixes were reviewed on commit `dd222e691238f6c3ac2fb2b41964d6d73d467d45`.

The scope for this audit includes and is limited to the following files:

- `contracts/contracts/validatorRegistry.algo.ts`: main contract, singleton.
- `contracts/contracts/stakingPool.algo.ts`: one instance deployed per `stakingPool`.
- `contracts/contracts/constants.algo.ts`: constants used across the contracts.

In the revision commit, type definitions were moved from `validatorRegistry.algo.ts` to a new contract:

- `contracts/contracts/validatorConfigs.algo.ts`: type definitions used across the contracts.

No other files in this repository were audited. Its dependencies are assumed to work according to their documentation. Also, no tests were reviewed for this audit.

[1] https://txnlab.gitbook.io/reti-open-pooling
[2] https://txnlab.gitbook.io/reti-open-pooling/technical-implementation/reti-node-daemon

Contracts Descriptions

contracts/contracts/validatorRegistry.algo.ts

The `ValidatorRegistry` is a singleton responsible for the following:

- Registering validators and approving, storing, and managing their settings
- Keeping track of each validator's global state
- Acting as a factory for `StakingPool` deployment on a Validator's request
- Offer a single on-chain endpoint to add stake to a requested validator and distribute the amount through its pools
- Maintaining the system's, validators', and stakers' global states
- Monitoring network state to perform Minimum Balance Requirement (MBR) and saturation calculations
- Allowing validators perform maintenance actions.

contracts/contracts/stakingPool.algo.ts

The `StakingPool` contract is deployed each time a validator wants to add a new pool to one of their nodes. It tracks users' stakes and rewards, updating the stake at each epoch distribution, thus enabling a compound staking scheme.

Each `StakingPool` contract is responsible for:

- Keeping track of the pool's state, pool's stakers, and pool's Annual Percentage Rate (APR)
- Updating Algo and token rewards and charging protocol fees through its public `epochBalanceUpdate()` function
- Providing stakers with endpoints to remove stake and claim tokens
- Offering validators endpoints to perform maintenance actions
- Serving as a vault for the reward token, if any, in the first pool of each validator's node

Methodology

CoinFabrik was provided with the source code, including automated tests that define the expected behavior, and general documentation about the project. Our auditors spent three weeks auditing the source code provided, which includes understanding the context of use, analyzing the boundaries of the expected behavior of each contract and function, understanding the implementation by the development team (including dependencies beyond the scope to be audited) and identifying possible situations in which the code allows the caller to reach a state that exposes some vulnerability. Without being limited to them, the audit process included the following analyses.

- Arithmetic errors

- Race conditions
- Misuse of block timestamps
- Denial of service attacks
- Excessive opcode budget usage
- Missing or misused function qualifiers
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling
- Insufficient validation of the input parameters
- Incorrect handling of cryptographic signatures
- Centralization and upgradeability

After delivering a report with our findings, the development team had the opportunity to comment on every finding and fix the issues they considered convenient. Once fixed and/or commented, our team ran a second review process to verify that the changes to the code effectively solve the issues found and do not unintentionally add new ones. This report includes the final status after the second review.

Findings

In the following table we summarize the security issues we found in this audit. The severity classification criteria and the status meaning are explained below. This table does not include the enhancements we suggest to implement, which are described in a specific section after the security issues.

ID	Title	Severity	Status
HI-01	Inconsistent Token Reward Calculation for Partial Stakers	High	Resolved
ME-01	Stake Entry Penalty Impacts on Previous Stakings	Medium	Acknowledged
ME-02	Incorrect Validator SunsettingOn Verification	Medium	Resolved
ME-03	Incentivizing Pool Saturation for Staker Gain	Medium	Resolved
ME-04	AVM Opcodes to be Implemented	Medium	Resolved

ID	Title	Severity	Status
MI-01	Inaccurate Reward Distribution When Only Partial Staking	Minor	Acknowledged
MI-02	Missed Rewards Due to Unexecuted Epoch Balance Update	Minor	Acknowledged
MI-03	Validator Manager not Funded if Saturated	Minor	Mitigated
MI-04	Weak Gating Mechanisms Based on Holding Specific Tokens	Minor	Resolved
MI-05	Missing Validations in Setter Methods for Validator Configuration	Minor	Resolved
MI-06	Incorrect Total Stake Check in addStake	Minor	Resolved

Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. Blocking bugs are also included in this category. They must be fixed **immediately**.
- **High:** These refer to a vulnerability that, if exploited, could have a substantial impact, but requires a more extensive setup or effort compared to critical issues. These pose a significant risk and **demand immediate attention**.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of, but might be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.

Issues Status

An issue detected by this audit has one of the following statuses:

- **Unresolved**: The issue has not been resolved.
- **Acknowledged**: The issue remains in the code, but is a result of an intentional decision. The reported risk is accepted by the development team.
- **Resolved**: Adjusted program implementation to eliminate the risk.
- **Partially resolved**: Adjusted program implementation to eliminate part of the risk. The other part remains in the code, but is a result of an intentional decision.
- **Mitigated**: Implemented actions to minimize the impact or likelihood of the risk.

Critical Severity Issues

No issues found.

High Severity Issues

HI-01 Inconsistent Token Reward Calculation for Partial Stakers

Location:

- `contracts/contracts/stakingPool.algo.ts:760-769`

The token reward calculation logic within the `epochBalanceUpdate` function contains an inconsistency that affects partial stakers. Specifically, when calculating rewards for partial stakers, the available token reward (`tokenRewardAvail`) is reduced on each iteration of the loop. This behavior leads to a situation where partial stakers with identical balances and time percentages may receive different rewards.

This issue arises because the reduction of `tokenRewardAvail` on each iteration causes the reward calculation for subsequent partial stakers to be based on an increasingly smaller pool of available tokens. As a result, partial stakers who are processed later in the loop may receive fewer tokens than those who were processed earlier, even if they have identical stakes and participation levels.

Recommendation

To address this issue, we recommend caching the initial value of `tokenRewardAvail` before entering the loop that calculates rewards for partial stakers. This cached value should be

used to calculate individual rewards in each iteration of the loop, while allowing the reduction of other variables as intended by the original code.

Status

Resolved. Fixed according to the recommendation.

Medium Severity Issues

ME-01 Stake Entry Penalty Impacts on Previous Stakings

Location:

- `contracts/contracts/stakingPool.algo.ts`

When a staker adds new stake to their existing stake using the `addStake()` function, an entry penalty is enforced by introducing a delay of 320 rounds during which the stake does not accrue rewards. In the staker's stake info, the `entryRound` value is set to the current round plus 320.

Due to the current implementation, the `epochBalanceUpdate()` function only considers the latest `entryRound` when calculating rewards. This means that any previous staking history is disregarded, and rewards are only accrued from the new `entryRound` onward. Consequently, stakers receive rewards only for the period between their most recent stake addition and the end of the epoch, even if they had been staking continuously before the new addition.

For example, in a scenario where both Alice and Bob started staking from the first round of a one-day-long epoch, with Alice having staked 1M ALGOs and Bob having staked 1 ALGO, if Alice adds more stake just before the epoch finishes (less than 320 rounds away), she will not be eligible for rewards, while Bob will receive all the rewards. In the next epoch, Alice total stake will accrue rewards again.

Recommendation

To avoid this issue, it is recommended to track every individual staking instance separately. If tracking of staking instances is not implemented, users should be aware that adding a new stake will reset their entry round and potentially affect their reward eligibility. This behavior should be clearly documented for users.

Status

Acknowledged. The development team stated tracking every stake would increase opcode costs and increase complexity. This behavior is already documented[3].

[3] https://txnlab.gitbook.io/reti-open-pooling/core-concepts/rewards#partial-epoch-staking

ME-02 Incorrect Validator SunsettingOn Verification

Location:

- `contracts/contracts/validatorRegistry.algo.ts:655`

The assert statement on line 655 of the system's code incorrectly verifies the `sunsettingOn` timestamp for validators in the system. This faulty verification logic leads to an incorrect implementation of staking restrictions.

When checking the `sunsettingOn` timestamp, the current condition is actually verifying the opposite condition of what is intended. As a result, staking is prevented before the `sunsettingOn` date has passed and allowed after it has been reached. This behavior is contrary to the system's intended logic.

Recommendation

Update the condition to correctly verify that staking is prohibited only after the `sunsettingOn` date has passed.

Status

Resolved. Fixed according to the recommendation.

ME-03 Incentivizing Pool Saturation for Staker Gain

Location:

- `contracts/contracts/stakingPool.algo.ts`

The current system creates an incentive for stakers to slightly saturate the pool, thereby skipping the validator's commission and instead directing the excess rewards to the fee sink in an amount lower than what would have been paid to the validator. This creates a situation where stakers can benefit personally by subtly manipulating the pool's saturation level.

This incentive from the stakers is adversarial to the incentive from the validators, who seek for their commission.

Recommendation

Eliminate this incentive by adjusting the reward distribution logic. Ensure the amount directed to the fee sink is always greater than the validator's commission.

Status

Resolved. Fixed according to the recommendation.

ME-04 AVM Opcodes to be Implemented

Location:

- `contracts/contracts/validatorRegistry.algo.ts: 1510, 1516`
- `contracts/contracts/stakingPool.algo.ts: 965, 980, 985, 993`

The TODO comments in the contracts which reference future AVM opcodes should be resolved and replace the hardcoded values. These opcodes are expected to provide improved functionality and flexibility. It is essential to implement these new opcodes before deploying to the mainnet to ensure consistency and take full advantage of the protocol's latest features, avoiding potential issues that could arise from outdated or hardcoded values.

Recommendation

Replace current hardcoded values with the new opcodes through TEALScript as soon as they are available.

Status

Resolved. Hardcoded values were replaced with new AVM opcodes through TEALScript.

Minor Severity Issues

MI-01 Inaccurate Reward Distribution When Only Partial Staking

Location:

- `contracts/contracts/stakingPool.algo.ts`

When there are only partial stakers in the system, the current implementation fails to distribute all allocated ALGOS (reward tokens) among them. This results in an excess of undistributed tokens being carried over to the next epoch.

These undistributed tokens will be distributed in the next epoch, but with different distribution parameters due to potential changes in pool participants and their participation times. In validators with short epoch times, this deviation from the theoretical reward distribution may be small. However, for very long epoch times the contrast could be significant.

Consider two stakers participating 50% of the time each, with a total of 200 ALGOS (100 ALGOS x 2). If there are 4 ALGOs in rewards, both stakers will receive 1 ALGO as reward, leaving 2 ALGOs undistributed. These 2 ALGOs will be distributed among all participants, including any new joiners, in the next epoch.

Recommendation

Implement a mechanism that ensures complete distribution of allocated ALGOS among all participants, even if they are partial stakers. If this is not feasible, users should be aware of this behavior.

Status

Acknowledged. The development team stated this is an expected result of the anti-gaming mechanism.

MI-02 Missed Rewards Due to Unexecuted Epoch Balance Update

Location:

- `contracts/contracts/stakingPool.algo.ts`

If the `epochBalanceUpdate()` function is not executed within an epoch (t), token rewards that accrued during the previous epoch (t-1) will not be distributed. This omission can lead to missed rewards for stakers, as the system does not account for backdated distributions.

Stakers may miss out on rewards that were earned during a previous epoch if the `epochBalanceUpdate()` function is not called within the current epoch. This can result in an accumulation of undistributed token rewards.

Recommendation

To avoid missed rewards, it is essential to execute the `epochBalanceUpdate()` function in every epoch. Stakers should be aware that if this function is not executed, token rewards will not be distributed, and they will miss out on potential earnings.

Status

Acknowledged. As the frontend tracks epoch balance updates, users will be notified through the UI when epochs are missed. Red dots are displayed to indicate missed epochs.

MI-03 Validator Manager not Funded if Saturated

Location:

- `contracts/contracts/stakingPool.algo.ts`

The `epochBalanceUpdate()` function checks the validator manager's balance to fund it when its balance is low, but this check is not performed when the validator is saturated. As a result, if the validator remains saturated for an extended period, the manager's balance

will not be checked and increased, potentially leaving them out of funds for staking operations.

This issue is considered minor because the validator could remove stakers to avoid saturation. However, this corrective action is not a desirable outcome. Removing stakers would be a last resort, as it would result in lost revenue for the removed staker and could damage the reputation of the validator.

It is worth noting that the incentives created by ME-03 directly contribute to this issue. By incentivizing stakers to saturate the pool, the system creates a situation where validator managers may not be funded for extended periods.

Recommendation

Modify the `epochBalanceUpdate()` function to always check and fund the validator manager's balance, regardless of whether the validator is saturated or not. This will ensure that the manager has sufficient funds for staking operations.

Status

Mitigated. Since ME-03 was resolved, stakers are incentivized to maintain the pool below the saturation level. Otherwise, their rewards would be reduced.

MI-04 Weak Gating Mechanisms Based on Holding Specific Tokens

Location:

- `contracts/contracts/validatorRegistry.algo.ts`

Classification:

- CWE-284: Improper Access Control[4]

Using gating mechanisms that depend on holding a specific token—whether fungible (ASA) or non-fungible (NFD)—is weak and potentially insecure. The requirement to hold the asset is only necessary at the moment of adding new staking, allowing users to bypass the intended gating mechanism by transferring the token between multiple addresses.

This weakness in the gating mechanism creates a security risk and undermines the exclusivity intended for token holders. Users can exploit this loophole by transferring the token to another address after adding staking, thereby gaining unregulated access to staking features.

[4] https://cwe.mitre.org/data/definitions/284.html

Recommendation

To strengthen the gating mechanism, document this weakness and recommend utilizing non-transferable tokens.

Status

Resolved. The development team informed us this gating check is now performed by the Reti node. Then, the manager account would remove those stakers from the pool.

MI-05 Missing Validations in Setter Methods for Validator Configuration

Location:

- `contracts/contracts/validatorRegistry.algo.ts`

Classification:

- CWE-20: Improper Input Validation[5]

The `ValidatorConfig` object is not properly validated when its properties are updated through setter methods. However, when a new validator is added using the `addValidator()` method, the entire `ValidatorConfig` object is validated using the `validateConfig()` method. This discrepancy in validation behavior between setter methods and the `addValidator()` method may result in configuration inconsistencies.

Hereunder is a list of the setter methods and their discrepancies:

changeValidatorManager(): Only checks if the transaction sender is the owner. Consider adding an assertion to check that the new manager's address is not a zero address.

```
assert(config.manager !== Address.zeroAddress)
```

changeValidatorSunsetInfo(): Only checks if the transaction sender is the owner. Consider adding an assertion to ensure that the `sunsettingOn` timestamp is greater than the current timestamp.

```
if (config.sunsettingOn !== 0) {
  assert(config.sunsettingOn > globals.latestTimestamp, 'sunsettingOn must be later than
now if set')
}
```

[5] https://cwe.mitre.org/data/definitions/20.html

changeValidatorNFD(): Verifies the NFD with an internal transaction but does not assert the result. Consider adding an assertion to check the result of the internal transaction that validates the NFD.

```
assert(btoi(this.itxn.lastLog) === 1, "provided NFD isn't valid")
```

changeValidatorRewardInfo(): Modifies gating and reward parameters without validation. Consider adding validations similar to those in the `validateConfig()` function to ensure new gating mechanisms and reward parameters are valid.

Recommendation

Implement the recommendations for each setter function previously listed.

Status

Resolved. Development team decided not to implement the assertion in `changeValidatorSunsetInfo()` because owners should be able to immediately stop their validator. The rest of the validations were implemented.

MI-06 Incorrect Total Stake Check in addStake

Location:

- `contracts/contracts/validatorRegistry.algo.ts: 672`

The assertion checking the total staked amount against the maximum allowed stake (`this.maxAllowedStake()`) is positioned incorrectly within the `addStake()` function. Currently, it is performed before updating the `totalAlgoStaked` variable, resulting in an inaccurate comparison. This can lead to a situation where the recorded total stake exceeds the defined `maxAllowedStake()`, potentially causing conflicts with other systems relying on accurate stake limits.

This is the staking check:

```
assert(
  this.validatorList(validatorId).value.state.totalAlgoStaked < this.maxAllowedStake(),
  'total staked for all of a validators pools may not exceed hard cap',
)
```

Recommendation

Move the assertion check immediately after updating the total staked amount.

Status

Resolved. Now the assertion considers the amount to be staked.

Enhancements

These items do not represent a security risk. They are best practices that we suggest implementing.

ID	Title	Status
EN-01	Resolve TODO Comments	Implemented

EN-01 Resolve TODO Comments

Location:

- `contracts/contracts/validatorRegistry.algo.ts: 196`
- `contracts/contracts/stakingPool.algo.ts: 95, 1037`

Outstanding "TODO" comments within the codebase represent unaddressed areas requiring further development or clarification. Leaving these unresolved can lead to an accumulation of technical debt, hindering future modifications and increasing the risk of introducing vulnerabilities. Additionally, these comments clutter the code, making it less readable and comprehensible.

Recommendation

Resolve “TODO” comment instances.

Status

Implemented.

Other Considerations

The considerations stated in this section are not right or wrong. We do not suggest any action to fix them. But we consider that they may be of interest to other stakeholders of the project, including users of the audited contracts, token holders or project investors.

Centralization

The system is designed to minimize centralization by implementing safeguards such as limits on the maximum stake a validator can control within a single pool and across all pools. This ensures that no single validator can dominate the network. Additionally, the open nature of validator participation, where anyone can set up a node and manage pools, promotes a decentralized network structure. The ability for any user to trigger essential

operations like epoch updates further reduces reliance on specific validators, distributing power and responsibility across the network.

Upgrades

The system supports upgradeability by allowing validator owners to modify key settings, such as management and commission addresses, and to sunset operations when needed.

The contracts include an upgrade mechanism intended solely for testing. This can only be triggered by a hardcoded address that exists exclusively in testing environments. This function is meant to be removed before deployment to the mainnet.

In commit `100bc93442c7c06ad29fe462e9ecba79f07102f8`, testing upgrade mechanisms were removed from the contracts.

Privileged Roles

These are the privileged roles that we identified on each of the audited contracts.

Validator Owner

The ultimate controller of a validator. They have authority over the validator's configuration, including setting the owner address, changing the management and commission addresses, and initiating sunsetting processes.

This role has full control over validator settings, including security changes like replacing the manager account if compromised.

Manager Account

A hot-wallet associated with a validator, responsible for executing operational commands such as issuing transactions for pool management, participation key creation, and epoch updates.

This role handles all day-to-day operations of the validator and signs transactions on its behalf.

TEALScript Production Readiness

Please be advised that the codebase for this project is written in TEALScript, which is currently a work in progress and not deemed production-ready.

Due to the developmental nature of TEALScript, there may be inherent risks and uncertainties associated with its use. These could include, but are not limited to,

unexpected behaviors, compiler issues, or other unforeseen problems that may not have been identified or addressed during this audit.

Changelog

- 2024-08-21 – Initial report based on commit `1e2564b2df94657d86b7730d2b2d87f136b278a1`.
- 2024-10-15 – Fixes checked on commit `dd222e691238f6c3ac2fb2b41964d6d73d467d45`.

Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the Réti project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.