

Ejection Functionality Smart Contract Review

Minimum Spanning Technologies

Version 1.0 – June 10, 2024

Prepared By
Parnian Alimi
Paul Bottinelli
Javed Samuel

Prepared For
Sam Williams
Ivan Morozov
Christian Smorra

1 Executive Summary

Synopsis

In June 2024, Arweave engaged NCC Group's Cryptography Services team to conduct a follow-up security assessment of the staking smart contract. This smart contract enables its owner to create (public or private) pools and modify them. Users can deposit tokens to these pools in order to earn stake and claim rewards. The codebase is a fork of the MorpheusAIs project which has been extensively audited[1]. The review was delivered by two consultants over five person-days.

Scope

NCC Group's evaluation focused on the *contracts/Distribution.sol* and *contracts/Refunder.sol* solidity files on commit `1041719`. The review specifically concentrated on two recent updates:

1. An update to the *Distribution* contract with the addition of an `ejectStakedFunds()` function allowing for the proactive refunding of individual users, and
2. An update to introduce the capability to proactively refund a batch of users through a Multicall mechanism (in *Refunder.sol*).

The testing methodology was manual code review, with a focus on identifying common smart contract and software security pitfalls.

Towards the end of the engagement some of NCC Group's feedback were addressed in commit `df8935e` and commit `0570915`.

Limitations

Good coverage of the in-scope code was achieved overall. However, note that the contracts were evaluated in isolation, but will be used as part of a more complex system which is not covered by this review. In addition, the implementation makes a few assumptions regarding trusted entities, including about the fallback address, the deposit token contract, and the Distribution contract owner. It is paramount for the security of the system that these trusted entities are indeed honest. Further discussions on the topic are included in Audit Notes.

Key Findings

The assessment uncovered a total of three findings.

- Finding "Contract's Total Funds May Reach Inconsistent State when `amountToTransfer > poolData.totalDeposited`" uncovers a transfer amount miscalculation that results in the eject function (`ejectStakedFunds()`) not being able to eject the entire remaining balance.
- Finding "Inconsistent Error Handling During Staking" identifies a minor inconsistency in error handling.
- Finding "Events may be Emitted Out of Order Due to Reentrancy" analyzes the impact of reentrancy type attacks in the *Distribution* contract.

Additionally, the Audit Notes section contains a number of more informal engagement notes as well as a discussion about trusted entities.

Strategic Recommendations

First, prioritize addressing the findings and numerous notes in the Audit Notes section. Second, the *Distribution* contract contains outdated code portions pertaining to the MorpheusAIs project, but which are no longer pertinent to Arweave's deployment. All

1. https://github.com/MorpheusAIs/Docs/tree/main/Security%20Audit%20Reports/Distribution%20Contract



obsolete logic (including the support for private pools) should be removed in order to reduce the attack surface as well as to reduce maintenance overhead.

Third, use a static code analysis tools, such as `npm audit`, to catch outdated or vulnerable versions of the dependency packages. Running `npm audit` on the current repository revealed 51 vulnerabilities (42 moderate, 8 high, 1 critical). Although some of these vulnerabilities might not be applicable to the smart contract's functionality, it is strongly recommended to keep abreast of the latest developments.



2 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Contract's Total Funds May Reach Inconsistent State when `amountToTransfer > poolData.totalDeposited`	New	VBW	Medium
Inconsistent Error Handling During Staking	New	LN7	Low
Events may be Emitted Out of Order Due to Reentrancy	New	YPF	Info



3 Finding Details

Medium

Contract's Total Funds May Reach Inconsistent State when `amountToTransfer > poolData.totalDeposited`

Overall Risk	Medium	**Finding ID**	NCC-E014939-VBW
Impact	High	**Component**	Distribution Contract
Exploitability	Medium	**Category**	Data Validation
		Status	New

Impact

The transfer of an incorrect amount of funds when the `amountToTransfer > poolData.totalDeposited` may result in some inconsistencies in the total fund supply.

Description

The newly introduced function `ejectStakedFunds()` in the file *Distribution.sol* is designed to allow for the proactive refunding of individual users. The function is excerpted below, for convenience.

This function expects a `poolId` as argument, as well as the address of the `user` it should refund. The amount to be refunded is obtained by getting the value of the `userData.deposited` field, as can be seen on line 362 below. The function then resets a few fields, and proceeds to "Reduce the total deposited amount in the pool data" on line 373 below.

After that, the code explicitly handles the case where `amountToTransfer` is strictly larger than the `poolData.totalDeposited` in the `else` clause highlighted below. That is, if the `amountToTransfer` is strictly larger than the `poolData.totalDeposited`, the function sets the pool's `totalDeposited` amount to 0 and proceeds. Note that the function performs a similar operation in the case of public pools a few lines below.

Importantly, this means that in case the `amountToTransfer` is strictly larger than the `poolData.totalDeposited`, setting `poolData.totalDeposited` to `0` means that the total deposited amount was reduced by `totalDeposited` and not by `amountToTransfer`.

However, when performing the token transfer on line 390 highlighted below, the function transfers an amount equal to `amountToTransfer`, regardless of whether the `amountToTransfer` was strictly larger than the `poolData.totalDeposited`. Hence, the function may try to perform a transfer with more funds that are available.

```
358     function ejectStakedFunds(uint256 poolId, address user) external onlyRefunder {
359         require(user != address(0), "DS: user is the zero address");
360
361         UserData storage userData = usersData[user][poolId];
362         uint256 amountToTransfer = userData.deposited;
363         if (amountToTransfer > 0) {
364             // Clear userData to prepare for transfer
365             userData.deposited = 0;
366             userData.pendingRewards = 0;
367             userData.rate = 0;
368
369             // Update pool and global state before transferring funds
```

```
370             PoolData storage poolData = poolsData[poolId];
371             Pool storage pool = pools[poolId];
372
373             // Reduce the total deposited amount in the pool data
374             if (poolData.totalDeposited >= amountToTransfer) {
375                 poolData.totalDeposited -= amountToTransfer;
376             } else {
377                 poolData.totalDeposited = 0; // In case of any discrepancies
378             }
379
380             // If the pool is public, update the global deposited total as well
381             if (pool.isPublic) {
382                 if (totalDepositedInPublicPools >= amountToTransfer) {
383                     totalDepositedInPublicPools -= amountToTransfer;
384                 } else {
385                     totalDepositedInPublicPools = 0; // In case of any discrepancies
386                 }
387             }
388
389             // Perform the transfer
390             IERC20(depositToken).safeTransfer(user, amountToTransfer);
391
392             // Emit successful withdrawal event
393             emit UserWithdrawn(poolId, user, amountToTransfer);
394         }
395     }
396 }
```

Figure 1: staking-smart-contracts/contracts/Distribution.sol

Although the call to `safeTransfer()` might fail and revert in case of insufficient balance, it seems that the discrepancy between the amount transferred in case `amountToTransfer` is strictly larger than the `poolData.totalDeposited` would result in some inconsistencies in the total fund supply and might prevent users from getting their remaining balance out.

Recommendation

Update the function logic to ensure that in case `amountToTransfer` is strictly larger than the `poolData.totalDeposited`, the `amountToTransfer` gets set to exactly what was decremented from `poolData.totalDeposited` on line 377, that is, set `amountToTransfer` to `poolData.totalDeposited`. Ensure the same logic applies to `totalDepositedInPublicPools`.

Location

staking-smart-contracts/contracts/Distribution.sol

Low

Inconsistent Error Handling During Staking

Overall Risk	Low	**Finding ID**	NCC-E014939-LN7
Impact	Low	**Component**	Distribution Contract
Exploitability	Low	**Category**	Data Validation
		Status	New

Impact

Inconsistent error handling leads to fragile state.

Description

In the following code snippet from the *Distribution* contract, when the inputted `amount_` is 0 the call is reverted with a “DS: nothing to stake” message. However, in case the actual transferred amount (calculated on line 221) is 0, the code proceeds to update the pool and user’s last updated and last staked timestamps. This behavior is inconsistent.

```
208 function _stake(address user_, uint256 poolId_, uint256 amount_, uint256 currentPoolRate_)
    ↳ private {
209     require(amount_ > 0, "DS: nothing to stake");
210
211     Pool storage pool = pools[poolId_];
212     PoolData storage poolData = poolsData[poolId_];
213     UserData storage userData = usersData[user_][poolId_];
214
215     if (pool.isPublic) {
216         // https://docs.lido.fi/guides/lido-tokens-integration-guide/#steth-internals-
            ↳ share-mechanics
217         uint256 balanceBefore_ = IERC20(depositToken).balanceOf(address(this));
218         IERC20(depositToken).safeTransferFrom(_msgSender(), address(this), amount_);
219         uint256 balanceAfter_ = IERC20(depositToken).balanceOf(address(this));
220
221         amount_ = balanceAfter_ - balanceBefore_;
222
223         require(userData.deposited + amount_ >= pool.minimalStake, "DS: amount too low");
224
225         totalDepositedInPublicPools += amount_;
226     }
227
228     userData.pendingRewards = _getCurrentUserReward(currentPoolRate_, userData);
229
230     // Update pool data
231     poolData.lastUpdate = uint128(block.timestamp);
232     poolData.rate = currentPoolRate_;
233     poolData.totalDeposited += amount_;
234
235     // Update user data
236     userData.lastStake = uint128(block.timestamp);
237     userData.rate = currentPoolRate_;
238     userData.deposited += amount_;
239
240     emit UserStaked(poolId_, user_, amount_);
241 }
```




Recommendation

Consider adding a check after line 221 to ensure that `amount_` is not 0.

Location

staking-smart-contracts/contracts/Distribution.sol



Info

Events may be Emitted Out of Order Due to Reentrancy

Overall Risk	Informational	**Finding ID**	NCC-E014939-YPF
Impact	Undetermined	**Component**	Distribution Contract
Exploitability	Undetermined	**Category**	Auditing and Logging
		Status	New

Impact

Emitting events after an external call can result in out-of-order events emission due to reentrancy. This can have adverse effects on a system that relies on events to initiate actions off-chain.

Description

Checks-effects-interactions is a Solidity best practice coding pattern[2], which prevents a malicious external call from causing damage by reentering the contract. It is recommended to structure the body of functions such that input validations (checks) are performed first and are followed by all the state changes (effects). Any required external functions should be called last (interactions).

Events are considered to be effects, depending on the greater off-chain architecture, and should be emitted before the external calls. Consider the following contract:

```
pragma solidity ^0.8.10;

contract OutOfOrderEventExample {
    event ImportantEvent1();
    event ImportantEvent2();

    function function1() public {
        // ...checks...
        // ...state changes...

        // External call to MaliciousContract before event emission
        call_malicious_contract()
        // determine success

        // Event is emitted after the external call
        emit ImportantEvent1();
    }

    function function2() public {
        // ...checks...
        // ...state changes...
        emit ImportantEvent2();
    }
}
```

A) An attacker that controls the *MaliciousContract* calls `function1()` and **B)** reenters the *OutOfOrderEventExample* contract's `function1()` (for the second time) or `function2()`. This

2. https://docs.soliditylang.org/en/latest/security-considerations.html#reentrancy



reentrancy will result in an `ImportantEvent1(B)` or `ImportantEvent2(B)` emission before the original call to `function1()` emits an `ImportantEvent1(A)`.

In the context of the *Distribution* contract, for instance, a call to `withdraw()` results in an external call to the `depositToken` contract which, if successful, is followed by a `UserWithdrawn` event. If the `depositToken` contract is malicious it can call *Distribution*'s `stake()` function in its fallback function[3], which emits a `UserStaked` event if successful. In this scenario the withdrawal event will appear after the staking event even though they were executed in the opposite order.

Besides emitting events before external calls, it is best practice to apply required state changes before external calls. Let's take a closer look at the `_stake()` function, depicted below, which transfers an `amount_` of `depositToken` from the caller/user into one of *Distribution* contract's pools. This implementation does not follow the recommended Checks-Effects-Interactions pattern, i.e. the contract's state is updated after the external call to the `depositToken` contract. A malicious `depositToken` contract can re-enter the *Distribution* contract during this call, but does not appear to be able to cause any damage, for instance, by manipulating the rewards calculation.

```
function _stake(address user_, uint256 poolId_, uint256 amount_, uint256 currentPoolRate_)
↳ private {
    require(amount_ > 0, "DS: nothing to stake");

    Pool storage pool = pools[poolId_];
    PoolData storage poolData = poolsData[poolId_];
    UserData storage userData = usersData[user_][poolId_];

    if (pool.isPublic) {
        // https://docs.lido.fi/guides/lido-tokens-integration-guide/#steth-internals-share-
        ↳ mechanics
        uint256 balanceBefore_ = IERC20(depositToken).balanceOf(address(this));
        IERC20(depositToken).safeTransferFrom(_msgSender(), address(this), amount_);
        uint256 balanceAfter_ = IERC20(depositToken).balanceOf(address(this));

        amount_ = balanceAfter_ - balanceBefore_;

        require(userData.deposited + amount_ >= pool.minimalStake, "DS: amount too low");

        totalDepositedInPublicPools += amount_;
    }

    userData.pendingRewards = _getCurrentUserReward(currentPoolRate_, userData);

    // Update pool data
    poolData.lastUpdate = uint128(block.timestamp);
    poolData.rate = currentPoolRate_;
    poolData.totalDeposited += amount_;

    // Update user data
    userData.lastStake = uint128(block.timestamp);
    userData.rate = currentPoolRate_;
    userData.deposited += amount_;

    emit UserStaked(poolId_, user_, amount_);
}
```

3. See https://solidity-by-example.org/hacks/re-entrancy/ for a minimal example.



This finding is considered to be of low severity, since the `depositToken` is set during *Distribution* contract's initialization and is assumed to be an honest ERC20 implementation.

Recommendation

Consider moving all event emissions before external calls.

Location

- staking-smart-contracts/contracts/Distribution.sol#L228
- staking-smart-contracts/contracts/Distribution.sol#L173



4 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.



Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.



5 Audit Notes

This section consists of notes and observations from the review that did not warrant standalone security findings, or that are not security related, but may nevertheless be of interest to the team at Arweave.

Observations on the `_withdraw()` Function

The following two notes pertain to the `_withdraw()` function in *contracts/Distribution.sol*.

Late Validation of `amount_` Parameter

The `_withdraw()` function checks that the amount to withdraw is strictly positive only very late in the function. Namely, it does so on line 271 in the source code and which we've also highlighted below.

```
function _withdraw(address user_, uint256 poolId_, uint256 amount_, uint256 currentPoolRate_)
↳ private {
    Pool storage pool = pools[poolId_];
    PoolData storage poolData = poolsData[poolId_];
    UserData storage userData = usersData[user_][poolId_];

    uint256 deposited_ = userData.deposited;
    require(deposited_ > 0, "DS: user isn't staked");

    if (amount_ > deposited_) {
        amount_ = deposited_;
    }

    uint256 newDeposited_;
    if (pool.isPublic) {
        // ...

        require(amount_ > 0, "DS: nothing to withdraw");
        require(newDeposited_ >= pool.minimalStake || newDeposited_ == 0, "DS: invalid
        ↳ withdraw amount");
    } else {
        newDeposited_ = deposited_ - amount_;
    }
```

In comparison, the `_stake()` function performs this check as a first step, see below.

```
function _stake(address user_, uint256 poolId_, uint256 amount_, uint256 currentPoolRate_)
↳ private {
    require(amount_ > 0, "DS: nothing to stake");
    // ...
```

Additionally, the `amount_` parameter is never checked to be greater than 0 in the case of a private pool, see the `else` clause of the above excerpt.

Undocumented Withdrawal Behaviour

The `_withdraw()` function enforces that any withdrawal action can either withdraw *all* the funds of an account, or an amount that does not result in the deposited funds of a user to go below the pool's `minimalStake`. This condition is enforced by the following `require` statement, which can also be spotted in the above excerpt of the `_withdraw()` function.

```
require(newDeposited_ >= pool.minimalStake || newDeposited_ == 0, "DS: invalid withdraw
↳ amount");
```

As a result, if a user withdraws an amount which results in their deposited funds to be in the range $[1, minimalStake - 1]$, then the function would not let them. This potentially



confusing behaviour does not appear to be documented anywhere. Consider the contract interface (located in *contracts/interfaces/IDistribution.sol*) which defines the `withdraw` function. The documentation does not mention the behaviour detailed above.

```
/**
* The function to withdraw tokens from the pool.
* @param poolId_ The pool's id.
* @param amount_ The amount of tokens to withdraw.
*/
function withdraw(uint256 poolId_, uint256 amount_) external;
```

Consider at the very least documenting this behaviour.

Contract Owner is Fully Trusted

Arweave's *Distribution* contract replaces upstream contract's *stEth* bridging mechanism with a wallet address. The Morpheus's *Distribution* contract utilizes an *L1Sender*[4] contract to:

1. bridge Distribution contract's overplus *stEth* tokens to the destination chain (Arbitrum) (via `sendDepositToken()`),
2. send a mint native tokens message to the destination chain when users claim rewards (via `sendMintMessage()`).

In place of the *L1Sender* role, Arweave's *Distribution* contract transfers overplus tokens to a wallet address (`aoDistributionWallet`). Also, note that the `claim()` API does not send a minting message to a destination chain or transfer the pending rewards to the user account; it will just simply emit an event. The `claim()` API has been later removed in commit `df8935e` to further simplify the contract.

Arweave team indicated that they are planning to renounce the contract owner role once the contract is deployed and is stable. The following 2 observations are only applicable for as long as the owner role exists. Both of these functionalities will be unreachable once the owner address is set to 0 through a call to the `renounceOwnership()` which is inherited from the base OwnableUpgradeable contract.

Pool Edits can Lead to Unfair Reward Payment

As is pointed out by the Code Hawks Public Report, the contract owner can edit a pool's `payoutStart` configuration to create an unfair payout scheme. Consider a scenario where a pool is set to start paying rewards starting at `payoutStart_1` timestamp (when `createPool()` is called), and the *Distribution* contract owner edits the pool (by calling `editPool()` API) and sets `payoutStart` to `payoutStart_2`:

- If `payoutStart_1 < payoutStart_2`: The rewards amount that is calculated by `LinearDistributionIntervalDecrease.getPeriodReward()` will decrease. This will decrease the pool's current rate, which in turn decreases the amount that the users can withdraw. This creates an unfair situation for the users who did not withdraw their rewards before the pool is edited.
- If `payoutStart_1 > payoutStart_2`: The users who withdrew their rewards before the pool is edited received a lower rewards amount than the users that will withdraw after the edit.

Similarly, the *Distribution* contract's admin can manipulate a pool's `decreaseInterval` (affects pool rate), `withdrawLockPeriod` (affects staked tokens' locking period), `claimLockPeriod` (affects staked tokens' locking period), `withdrawLockPeriodAfterStake` (affects staked

4. https://github.com/MorpheusAIs/Docs/blob/main/Smart%20Contracts/L1Sender.md

tokens' locking period), `initialReward` (affects the rewards amount), `rewardDecrease` (affects the rewards amount), and `minimalStake` (affects the enforced minimum stake amount) parameters. The owner's ability to easily manipulate the rewards' rate can deter users from participating in staking. Following NCC Group's recommendation, the `editPool()` function has been removed in commit `df8935e`.

Batch Stake/Withdraw Calls has a Different Effect than Individual Calls

Calls to `_stake()` and `_withdraw()` internal functions will update the pool's reward rate by updating the `lastUpdate` timestamp and the `totalDeposited` amount. The `manageUsersInPrivatePool()` API calculates the pool's current rate once and uses it for all `stake()` and `withdraw()` calls in the batch. Thereby when the owner calls `manageUsersInPrivatePool()` it will yield a different state than when users call `stake()` and `withdraw()` themselves. Following NCC Group's recommendation, the `manageUsersInPrivatePool()` function has been removed in commit `df8935e`.



6 Contact Info

The team from NCC Group has the following primary members:

- Paul Bottinelli – Consultant
  paul.bottinelli@nccgroup.com
- Parnian Alimi – Consultant
  parnian.alimi@nccgroup.com
- Javed Samuel – Practice Director, Cryptography Services
  javed.samuel@nccgroup.com

The team from Minimum Spanning Technologies has the following primary members:

- Sam Williams
  sam@arweave.org
- Ivan Morozov
  ivan@dataos.so

