Arcade Staking Rewards Audit Report

Version 2.0

Audited by:

MiloTruck

alexxander

March 18, 2024

Contents

1 Introduction

1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

More of our work can be found here.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an 'as-is' and 'as-available' basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol’s core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol’s functionality
- Low - Funds are **not** at risk

1.3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

2 Executive Summary

2.1 About Arcade Staking Rewards

ArcadeStakingRewards is inspired by the Synthetix StakingRewards system. The contract facilitates a staking mechanism where users can stake ERC20 tokens `stakingToken` to earn rewards over time in the form of another ERC20 token `rewardsToken`. The rewards depend on the amount staked and the duration of the stake. Features of the protocol include:

- **Multiple Deposits**: Users can make multiple deposits, with each deposit accruing rewards independently until the staking period ends.
- **Lock Period**: Upon staking, users must commit to a fixed lock period, during which funds are immovable. Early withdrawal before this period ends is not permitted.
- **Automatic Re-staking**: Post-lock period, funds automatically enter the next staking cycle without a lock period, allowing for withdrawal at any time.
- **Bonus Multipliers**: The contract offers bonus multipliers based on the chosen lock period (short, medium, or long), incentivizing longer commitments.
- **Voting Power**: The contract gives users governance capabilities by serving as a voting vault. When users stake, they gain voting power, which can be used in ArcadeDAO governance.

2.2 Overview

Project	Arcade Staking Rewards
Repository	dao-contracts
Commit Hash	4c259b440795...
Mitigation Hash	333875042be1...
Date	8 March 2024 - 11 March 2024

2.3 Issues Found

Severity	**Count**
High Risk	1
Medium Risk	2
Low Risk	3
Informational	3
Total Issues	**9**

3 Findings Summary

ID	Description	Status
H-1	Attacker can remove voting power from more than 52 minutes ago due to maximum queue length	Resolved
M-1	Missing `whenNotPaused` modifier for `exitAll()`	Resolved
M-2	Rewards are lost when `notifyRewardAmount()` is called multiple times before the first staker	Resolved
L-1	Unsafe `uint96` casts in `_addVotingPower()` and `_subtractVotingPower()`	Resolved
L-2	`reward > balance` check in `_startRewardEmission()` is not comprehensive	Resolved
L-3	Subtracting the remainder from `reward` in `notifyRewardAmount()` reduces rewards unnecessarily	Resolved
I-1	Use a constructor to initialise `rewardsDistribution` in `ArcadeRewardsRecipient.sol`	Resolved
I-2	Code in `getActiveStakes()` can be simplified	Resolved
I-3	Inconsistent emission of `RewardPaid` events	Resolved

4 Findings

High Risk

[H-1] Attacker can remove voting power from more than 52 minutes ago due to maximum queue length

Context:

- ArcadeStakingRewards.sol#L122-L124
- ArcadeStakingRewards.sol#L1025-L1026
- BoundedHistory.sol#L92-L96

Description: The voting power queue of each user has a maximum length of `MAX_HISTORY_LENGTH`. In the current implementation, the queue can have a maximum length of 256:

```
// Max length of any voting history. Prevents gas exhaustion
// attacks from having too-large history.
uint256 public constant MAX_HISTORY_LENGTH = 256;
```

For example, when pushing a new checkpoint into the voting power queue in `changeDelegation()`, `MAX_HISTORY_LENGTH` is passed to `push()`:

```
// Store the increase in power
votingPower.push(newDelegate, newDelegateVotes + userBalance, MAX_HISTORY_LENGTH);
```

In `BoundedHistory.push()`, a new checkpoint with the current block number is added to the front of the queue. Afterwards, if the queue's new length exceeds 256, the oldest checkpoint is deleted from the back of the queue:

```
} else if (length - minIndex >= maxLength) {
    // We need to push to the array, but if array is full to maxLength, so
    // we clear the oldest entry and increment the minIndex
    _clear(minIndex, ++minIndex, storageData);
}
```

However, such an implementation allows an attacker to force a user's history to only store the last `MAX_HISTORY_LENGTH` blocks. Since the maximum queue length is 256 in the current implementation, an attacker can force the contract to only store a user's voting power history for the last 256 blocks.

This can be achieved by:

- At every block, call `changeDelegation()` with `newDelegate` set to the victim's address. This will push a new entry into `votingPower` for the current block number.
- Due to the 256-entry limit for `votingPower`, after `changeDelegation()` is called enough times for `votingPower` to hold more than 256 entries, `push()` will start to delete the oldest entry.
- After 256 blocks, all past entries in `votingPower` will have been deleted, so the oldest entry will be from 256 blocks ago.

If queryVotePower() is called to query a user's voting power more than 256 blocks ago, the function will revert as there is no record of the user's voting power before or during blockNumber.

Since mainnet produces a block every 12 seconds, 256 blocks corresponds to 51.2 minutes, which means an attacker can forcefully remove a victim's voting power history from more than ~52 minutes ago.

Note that deposit() and withdraw() can also be used to perform the same attack, since they also add checkpoints to the delegate's queue with _addVotingPower() and _subtractVotingPower().

Recommendation: Consider removing the MAX_HISTORY_LENGTH limit on the checkpoint queue (i.e. the queue can be infinitely long). This can be achieved by setting MAX_HISTORY_LENGTH to an extremely large value, such as type(uint256).max, or reverting the contract to use History instead of BoundedHistory.

To prevent the issue of queryVotePower() running out of gas when the queue is too long, avoid clearing stale blocks in the queue when queryVotePower() is called:

```
  function queryVotePower(
      address user,
      uint256 blockNumber,
      bytes calldata
  ) external override returns (uint256) {
-     // Get our reference to historical data
-     BoundedHistory.HistoricalBalances memory votingPower = _votingPower();
-     // Find the historical data and clear everything more than 'staleBlockLag' into
the past
-     return
-         votingPower.findAndClear(
-             user,
-             blockNumber,
-             block.number - STALE_BLOCK_LAG
-         );
+     return this.queryVotePowerView(user, blockNumber);
  }
```

This would be similar to other implementations of voting checkpoint queues, such as OpenZeppelin's Votes.sol, which only adds checkpoints to the queue and but never removes them.

The only functions remaining that iterate through the checkpoint queue would be queryVotePower() and queryVotePowerView(), which performs a maximum of $log(n)$ iterations using binary search. Therefore, there isn't a need to remove stale checkpoints from the queue as there is no risk of any function ever running out-of-gas, resulting in DOS.

Arcade: Fixed in PR-45.

Renascence: Verified, the contract was reverted to use History instead of BoundedHistory. queryVotePowerView() also was changed to external and used in queryVotePower() instead of performing an external call. Additionally, the STALE_BLOCK_LAG variable was removed as well since it is no longer needed.

Medium Risk

[M-1] Missing `whenNotPaused` modifier for `exitAll()`

Context:

- ArcadeStakingRewards.sol#L555
- ArcadeStakingRewards.sol#L596

Description: `withdraw()` has the `whenNotPaused` modifier:

```
function withdraw(uint256 amount, uint256 depositId) public whenNotPaused
nonReentrant updateReward {
```

Therefore, when the contract is paused, users should not be able to withdraw their deposited tokens.

However, the `exitAll()` function, which users can call to withdraw tokens from all their deposited positions, does not have the `whenNotPaused` modifier:

```
function exitAll() external nonReentrant updateReward {
```

This is inconsistent with `withdraw()` since it wrongly allows users to withdraw even when the contract is paused.

Recommendation: Add the missing `whenNotPaused` modifier to `exitAll()`:

```diff
- function exitAll() external nonReentrant updateReward {
+ function exitAll() external whenNotPaused nonReentrant updateReward {
```

Arcade: Fixed in PR-42.

Renascence: Verified, `exitAll()` now has the `whenNotPaused` modifier.

[M-2] Rewards are lost when `notifyRewardAmount()` is called multiple times before the first staker

Context:

- ArcadeStakingRewards.sol#L673-L675

Description: In `notifyRewardAmount()`, if there is currently no deposits in the contract (i.e. `totalDeposits == 0`), `reward` is assigned to `notifiedRewardAmount`:

```
} else {
    notifiedRewardAmount = reward;
}
```

However, if `notifyRewardAmount()` is called multiple times before the first staker, `notifiedRewardAmount` will simply be overwritten with the new `reward` amount. As such, rewards from preceding calls to `notifyRewardAmount()` will be lost since they are no longer tracked.

Recommendation: Add `reward` to `notifiedRewardAmount` instead:

```diff
- notifiedRewardAmount = reward;
+ notifiedRewardAmount += reward;
```

Arcade: Fixed in PR-43.

Renascence: Verified, the recommended fix was implemented.

Low Risk

[L-1] Unsafe `uint96` casts in `_addVotingPower()` and `_subtractVotingPower()`

Context:

- ArcadeStakingRewards.sol#L903-L904
- ArcadeStakingRewards.sol#L856-L858

Description: In `_addVotingPower()` and `_subtractVotingPower()`, `amount`, which is the amount of voting power corresponding to the user's deposited token amount, is cast to `uint96` unsafely:

```
// Now we increase the user's balance
userData.amount += uint96(amount);
```

```
// Reduce the user's stored balance
// If properly optimized this block should result in 1 sload 1 store
userData.amount -= uint96(amount);
```

With the ARCD's current total supply of `1e26` it is unlikely for `amount` to overflow. However, if the total supply of ARCD is increased significantly in the future, it might become possible for `amount` to exceed `uint96`, causing the unsafe casts shown above to overflow.

Recommendation: Consider using safe casts instead, such as OpenZeppelin's `SafeCast.toUint96()`, or manually check that `amount` does not exceed `uint96` max:

```
if (amount > type(uint96).max) revert ASR_AmountTooBig("delegation");
```

Arcade: Fixed in PR-40.

Renascence: Verified, OpenZeppelin's `SafeCast.toUint96()` is now used.

[L-2] `reward > balance` check in `_startRewardEmission()` is not comprehensive

Context:

- ArcadeStakingRewards.sol#L752-L758

Description: `_startRewardEmission()` checks that `reward`, which is the amount of new reward tokens to distribute, is smaller than the contract's current reward token balance:

```
// Ensure the provided reward amount is not more than the balance in the contract.
// This keeps the reward rate in the right range, preventing overflows due to
// very high values of rewardRate in the earned and rewardsPerToken functions;
// Reward + leftover must be less than 2^256 / 10^18 to avoid overflow.
uint256 balance = rewardsToken.balanceOf(address(this));

if (reward > balance) revert ASR_RewardTooHigh();
```

However, this check is not comprehensive - `balance` includes leftover rewards from the previous period and unclaimed rewards from users. Therefore, it is possible for the contract to have insufficient reward tokens to distribute even when the `reward > balance` check passes.

For example, if `notifyRewardAmount()` is called with `reward = balance` and some rewards have not been claimed from the previous period, this check would pass but there would be insufficient reward tokens to distribute in the new reward period.

Recommendation: Track the total amount of unclaimed rewards from previous periods. Afterwards, modify the check to add leftover and unclaimed rewards to `reward`. This can be done as follows:

1. Add an `unclaimedRewards` state variable:

```
uint256 public unclaimedRewards;
```

2. `unclaimedRewards` stores the cumulative sum of `durationRewards` (i.e. the cumulative amount of rewards emitted):

```
- function rewardPerToken() public view returns (uint256) {
+ function rewardPerToken() public view returns (uint256, uint256) {
      if (totalDepositsWithBonus == 0) {
-         return rewardPerTokenStored;
+         return (rewardPerTokenStored, unclaimedRewards);
      }

      uint256 timePassed = lastTimeRewardApplicable() - lastUpdateTime;
      uint256 durationRewards = timePassed * rewardRate;
      uint256 updatedRewardsPerToken = (durationRewards * ONE) /
      totalDepositsWithBonus;

-     return rewardPerTokenStored + updatedRewardsPerToken;
+     return (rewardPerTokenStored + updatedRewardsPerToken, unclaimedRewards +
durationRewards);
  }
```

```
  modifier updateReward {
-     rewardPerTokenStored = rewardPerToken();
+     (rewardPerTokenStored, unclaimedRewards) = rewardPerToken();
      lastUpdateTime = lastTimeRewardApplicable();
      _;
  }
```

3. Note that a few other functions would need to be refactored as well to accommodate this change, since `rewardPerToken()` now returns two variables.

4. When users claim their rewards using `withdraw()`, `exitAll()`, `claimReward()` or `claimRewardAll()`, subtract the claimed amount from `unclaimedRewards`.

5. In `_startRewardEmission()`, modify the check as such:

```
+ uint256 leftover;
  if (block.timestamp >= periodFinish) {
      rewardRate = reward / rewardsDuration;
  } else {
      uint256 remaining = periodFinish - block.timestamp;
-     uint256 leftover = remaining * rewardRate;
+     leftover = remaining * rewardRate;
      rewardRate = (reward + leftover) / rewardsDuration;
  }

  // ...
  uint256 balance = rewardsToken.balanceOf(address(this));

- if (reward > balance) revert ASR_RewardTooHigh();
+ if (reward + leftover + unclaimedRewards > balance) revert ASR_RewardTooHigh();
```

Arcade: Fixed in PR-44.

Renascence: Verified, the recommended fix was implemented.

[L-3] Subtracting the remainder from `reward` in `notifyRewardAmount()` reduces rewards unnecessarily

Context:

- ArcadeStakingRewards.sol#L663-L669

Description: In `notifyRewardAmount()`, `reward` is reduced to be exactly divisible by `rewardsDuration`:

```
// check that the reward is divisible by the rewardsDuration
// to avoid rounding errors
uint256 remainder = reward % rewardsDuration;

if (remainder > 0) {
        reward -= remainder;
}
```

Afterwards, the function can take three possible paths:

1. If `totalDeposits > 0` and `block.timestamp >= periodFinish`, `reward` is divided by `rewardsDuration` in `_startRewardEmission()`:

```
if (block.timestamp >= periodFinish) {
    rewardRate = reward / rewardsDuration;
} else {
```

2. If `totalDeposits > 0` and `block.timestamp < periodFinish`, `reward + leftover` is divided by `rewardsDuration` instead, not `reward`:

```
} else {
    uint256 remaining = periodFinish - block.timestamp;
    uint256 leftover = remaining * rewardRate;
    rewardRate = (reward + leftover) / rewardsDuration;
}
```

3. If `totalDeposits == 0`, `reward` is simply assigned to `notifiedRewardAmount` and no division takes place.

Since paths 2 and 3 do not divide `reward` by `rewardsDuration`, subtracting the remainder from `reward` in `notifyRewardAmount()` unnecessarily reduces the contract's rewards when either path 2 or 3 is taken. For example:

- Assume the following:
  - `rewardsDuration` is 10 million seconds (approximately ~4 months).
  - No user has staked yet.
- `notifyRewardAmount()` is called with `reward` as 15 million base units of ARCD:
  - `reward % rewardsDuration` is 5 million, so `reward` is reduced to 10 million.
  - Since `totalDeposits == 0`, no division takes place and `reward` is simply assigned to `notifiedRewardAmount`.
- Therefore, 5 million ARCD has been lost unnecessarily.

Recommendation: Subtract the remainder in `_startRewardEmission()` instead of `notifyRewardAmount()`:

```
  function _startRewardEmission(uint256 reward) private {
      if (block.timestamp >= periodFinish) {
+         uint256 remainder = reward % rewardsDuration;
+         reward -= remainder;
          rewardRate = reward / rewardsDuration;
      } else {
```

Arcade: Fixed in PR-41.

Renascence: Verified, subtraction of the remainder was also needed in the `else` clause. Below is the revised and simplified version of the fix.

```
@@ -738,21 +736,27 @@ contract ArcadeStakingRewards is IArcadeStakingRewards,
ArcadeRewardsRecipient,
    * @param reward                          The amount of reward tokens to
    distribute.
    */
    function _startRewardEmission(uint256 reward) private {
-       if (block.timestamp >= periodFinish) {
-           rewardRate = reward / rewardsDuration;
-       } else {
+           uint256 leftover;
+
+       if (block.timestamp < periodFinish) {
            uint256 remaining = periodFinish - block.timestamp;
-           uint256 leftover = remaining * rewardRate;
-           rewardRate = (reward + leftover) / rewardsDuration;
+           leftover = remaining * rewardRate;
+
+           reward += leftover;
        }

+       uint256 remainder = reward % rewardsDuration;
+       reward -= remainder;
+
+       rewardRate = reward / rewardsDuration;
@@ -657,14 +663,6 @@ contract ArcadeStakingRewards is IArcadeStakingRewards,
ArcadeRewardsRecipient,
    function notifyRewardAmount(uint256 reward) external override whenNotPaused
    onlyRewardsDistribution updateReward {
        if (reward < ONE) revert ASR_MinimumRewardAmount();

-       // check that the reward is divisible by the rewardsDuration
-       // to avoid rounding errors
-       uint256 remainder = reward % rewardsDuration;
-
-       if (remainder > 0) {
-           reward -= remainder;
-       }
```

Informational

[I-1] Use a constructor to initialise `rewardsDistribution` in `ArcadeRewardsRecipient.sol`

Context:

- ArcadeStakingRewards.sol#L183

Description: On deployment, `rewardsDistribution` is directly assigned in the constructor of `ArcadeStakingRewards`:

```
rewardsDistribution = _rewardsDistribution;
```

However, it is a good practice to assign state variables belonging to an inherited contract in the contract's own constructor. In this case, `rewardsDistribution` should be initialised in the constructor of `ArcadeRewardsRecipient`.

Recommendation: Add a constructor in `ArcadeRewardsRecipient` to set `rewardsDistribution`.

In `ArcadeRewardsRecipient.sol`:

```
constructor(address _rewardsDistribution) {
    rewardsDistribution = _rewardsDistribution;
}
```

In `ArcadeStakingRewards.sol`:

```
  constructor(
      address _owner,
      address _rewardsDistribution,
      address _rewardsToken,
      address _arcdWethLP,
      uint256 _lpToArcdRate,
      uint256 _staleBlockLag
- ) Ownable(_owner) {
+ ) Ownable(_owner) ArcadeRewardsRecipient(_rewardsDistribution) {
```

The benefit of this approach is that if `ArcadeRewardsRecipient` is used in the future for other contracts, developers will not forget to initialize `rewardsDistribution`.

Arcade: Fixed in PR-37.

Renascence: Verified, `ArcadeRewardsRecipient` now contains a constructor that initializes `rewardsDistribution`, and `ArcadeStakingRewards` has been modified to accommodate this change. The zero address check for `_rewardsDistribution` was also moved into the constructor of `ArcadeRewardsRecipient`.

[I-2] Code in `getActiveStakes()` can be simplified

Context:

- ArcadeStakingRewards.sol#L341-L342

Description: The following code in `getActiveStakes()`:

```
activeStakes[activeIndex] = i;
activeIndex++;
```

can be simplified as such:

```
- activeStakes[activeIndex] = i;
- activeIndex++;
+ activeStakes[activeIndex++] = i;
```

Arcade: Fixed in PR-38.

Renascence: Verified, the recommended fix was implemented.

[I-3] Inconsistent emission of `RewardPaid` events

Context:

- ArcadeStakingRewards.sol

Description: The functions `claimRewardAll()` and `exitAll()` emit the event `RewardPaid(, , uint256 depositId)` for the corresponding `depositId` that is being processed.

```
# ArcadeStakingRewards.sol

function _processReward(UserStake storage userStake, uint256 reward) internal {
    if (reward > 0) {
        ...
>       emit RewardPaid(msg.sender, reward, stakes[msg.sender].length - 1);
    }
}
```

```
# ArcadeStakingRewards.sol

function claimRewardAll() external nonReentrant updateReward {
    ...
    for (uint256 i = 0; i < numUserStakes; ++i) {
        ...
        if (reward > 0) {
            ...
>           emit RewardPaid(msg.sender, reward, i);
        }
    ...
}
```

However, `claimReward()` and `withdraw()` call `_processReward()`, which always emits the event with `depositId` as the last element in the user's `UserStake[]` array, even if the processed `depositId` is not the last.

Recommendation: Pass `depositId` to `_processReward()`.

```
@@ -771,12 +772,11 @@ contract ArcadeStakingRewards is IArcadeStakingRewards,
ArcadeRewardsRecipient,
-   function _processReward(UserStake storage userStake, uint256 reward) internal {
+   function _processReward(UserStake storage userStake, uint256 reward, uint256
depositId) internal {
        if (reward > 0) {
            userStake.rewardPerTokenPaid = rewardPerTokenStored;
            rewardsToken.safeTransfer(msg.sender, reward);
-
-           emit RewardPaid(msg.sender, reward, stakes[msg.sender].length - 1);
+           emit RewardPaid(msg.sender, reward, depositId);
        }
    }
```

```
@@ -517,7 +518,7 @@ contract ArcadeStakingRewards is IArcadeStakingRewards,
ArcadeRewardsRecipient,
-   _processReward(userStake, reward);
+   _processReward(userStake, reward, depositId);
@@ -572,7 +573,7 @@ contract ArcadeStakingRewards is IArcadeStakingRewards,
ArcadeRewardsRecipient,
-   _processReward(userStake, reward);
+   _processReward(userStake, reward, depositId);
```

Arcade: Fixed in PR-39.

Renascence: Verified, the fix in the PR removes the `_processReward()` function and handles reward payouts similar to `claimRewardAll()` and `exitAll()`.