code4rena

Bitmind

Smart Contract Security Assessment

Version 2.0

Audit dates: Jan 22 — Jan 24, 2025

Audited by: 0xladboy
kupiasec

Contents

1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

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

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About Bitmind

Seraph ($SERAPH) is developing decentralized AI detection with Virtuals + Bittensor integration. With discrete staking mechanism in development (Base $SERAPH, $stTAO) for rewards from Agent revenues and expanding DeFAI partnerships, Seraph is pushing the boundaries on Authenticity Analysis of AI Agents (first usecase) and other Bittensor subnets integrations (later)

2.2 Scope

Repository	SeraphAgent/seraph-staking-contracts
Commit Hash	03b199e3115bc329c875a2d724777cf8e2e0a3ce
Mitigation Hash	3acac208cc14e98a4b0cd687d3adb5b3b736d973

2.3 Audit Timeline

DATE	EVENT
Jan 22, 2025	Audit start
Jan 24, 2025	Audit end
Jan 24, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	3
Low Risk	5
Informational	0
Total Issues	8

3. Findings Summary

ID	DESCRIPTION	STATUS
M-1	Re-adding a previously removed reward token leads to inaccurate reward amounts.	Resolved

M-2	One underfunded reward token locks all users' token reward	Resolved
M-3	Precision loss when `updateReward` truncates the rewardIndex	Resolved
L-1	`_isRewardTokenAllowed()` is not gas-efficient.	Resolved
L-2	Lock time isn't factored into rewards	Acknowledged
L-3	Admin can remove staking token via `recoverERC20`	Resolved
L-4	Missing min and max lock period validation when setting the _minLockPeriod	Resolved
L-5	Unsafe ERC20 Transfer Operations	Resolved

4. Findings

4.1 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] Re-adding a previously removed reward token leads to inaccurate reward amounts.

Severity: Medium	Status: Resolved

Target

- SeraphPool.sol

Severity:

- Impact: High
- Likelihood: Low

Description: Removing a reward token does not delete the associated reward index data. Consequently, the global reward index for the removed reward token remains intact. If the owner adds the removed reward token back, it may lead to incorrect reward calculations due to the preserved global index. Consider the following scenario:

- The owner removes `rewardToken_A`, but the global reward index for `rewardToken_A` remains undeleted.
- A new user stakes, resulting in a non-zero `balanceOf`, while their own reward index for `rewardToken_A` is 0.
- The owner then adds `rewardToken_A` back.
- When updating the user, their `earned[rewardToken_A]` will be calculated based on the difference between the global index and the user's own index. As a result, the earned value may be non-zero, even though the user did not actually earn any `rewardToken_A`.

Recommendation: Delete all reward index data. This may require modifying the structure of the `rewardIndexOf` mapping:

- From `rewardIndexOf[user][_rewardToken]` to `rewardIndexOf[_rewardToken][user]` throughout the entire codebase.
- This way, using `delete rewardIndexOf[_rewardToken];` will remove all users' reward index data.
- Additionally, `delete rewardIndex[_rewardToken];` will remove the global reward index.

Bitmind: Addressed with @f9baaab6bf90.. & @934f0e067f4..

Zenith: Verified.

[M-2] One underfunded reward token locks all users' token reward

Severity: Medium	Status: Resolved

Target

- SeraphPool.sol#claim

Severity:

- Impact: Medium
- Likelihood: Medium

Description: Within the `claim()` function, for each allowed reward token:

```
uint256 reward = earned[msg.sender][rewardToken];
if (reward > 0) {
    if (IERC20(rewardToken).balanceOf(address(this)) < reward) {
        revert SeraphPool__RewardTokenNotFound();
    }
    ...
}
```

- If the contract does not have enough of **one** particular reward token, it reverts, thereby **blocking** the claims for **all** other reward tokens.
- Users effectively lose access to the rest of their valid, claimable rewards.

This is inconvenient or even unfair if multiple reward tokens have different funding conditions or liquidity events. One underfunded token effectively "locks" the claim of all tokens.

Recommendation:

Introduce a Per-Token Claim Function
Provide a function like `claim(address _rewardToken)` that targets only one token at a time. This way, if Token A is out of balance, the user can still claim Token B without error. An example could be:

```
function claimSingleToken(address _rewardToken) external nonReentrant
whenNotPaused {
    _updateRewards(msg.sender);

    uint256 reward = earned[msg.sender][_rewardToken];
    if (reward > 0) {
```

```
        // Clear out earned
        earned[msg.sender][_rewardToken] = 0;

        // Ensure contract holds enough for payout
        if (IERC20(_rewardToken).balanceOf(address(this)) < reward) {
            revert SeraphPool__RewardTokenNotFound();
        }

        // Transfer reward
        rewardTotalSupply[_rewardToken] -= reward;
        IERC20(_rewardToken).transfer(msg.sender, reward);

        emit RewardClaimed(msg.sender, _rewardToken, reward);
    }
}
```

Bitmind: Addressed with the following commit - @e95cd09e9f4b..

Zenith: Verified.

[M-3] Precision loss when `updateReward` truncates the rewardIndex

Severity: Medium	Status: Resolved

Target

- SeraphPool.sol#updateRewardIndex

Severity:

- Impact: High
- Likelihood: Medium

Description:

```
    function updateRewardIndex(address _rewardToken, uint256
_rewardAmount) external onlyOwner {
        if (!_isRewardTokenAllowed(_rewardToken)) revert
SeraphPool__RewardTokenNotAllowed();
        if (totalSupply == 0) revert SeraphPool__NoStakedTokens();

        rewardTotalSupply[_rewardToken] += _rewardAmount;
        IERC20(_rewardToken).transferFrom(msg.sender, address(this),
_rewardAmount);
        rewardIndex[_rewardToken] += (_rewardAmount * MULTIPLIER) /
totalSupply;

        emit RewardIndexUpdated(_rewardToken, _rewardAmount);
    }
```

The reward token is updated from the computation:

```
        rewardIndex[_rewardToken] += (_rewardAmount * MULTIPLIER) /
totalSupply;
```

Assume the reward token is 6 decimals, but the staking token is 18 decimals.

- Numerator = 1e9 (reward) * 1e18 (multiplier) = 1e27.
- Denominator = totalSupply = 1e28.

The rewardIndex[_rewardToken] is round down to 0 even the 1e9 unit of reward token is added.

Precision loss when updateReward truncates the rewardIndex, especially when reward token has low decimals but the staking token has high decimals.

Recommendation: Use large MULTIPLIER to avoid precision loss.

```
uint256 private constant MULTIPLIER = 1e36;
```

and validate the staking token and reward token decimals to ensure both are 18 decimals.

Bitmind: Addressed with the following commit

Zenith: Verified.

4.2 Low Risk

A total of 5 low risk findings were identified.

[L-1] `_isRewardTokenAllowed()` is not gas-efficient.

Severity: Low	Status: Resolved

Target

- SeraphPool.sol

Severity:

- Impact: Low
- Likelihood: High

Description: The `_isRewardTokenAllowed()` function currently checks if a reward token is allowed by iterating through the `allowedRewardTokens` array. This approach is not gas-efficient.

```
    function _isRewardTokenAllowed(address _rewardToken) private view
returns (bool) {///? introduce mapping, for gas saving
        for (uint256 i = 0; i < allowedRewardTokens.length; i++) {
            if (allowedRewardTokens[i] == _rewardToken) {
                return true;
            }
        }
        return false;
    }
```

Recommendation: Introduce a new mapping (address => bool) to efficiently track all allowed reward tokens. This will enhance gas efficiency by eliminating the need for iteration.

Bitmind: Addressed with the following commit

Zenith: Verified.

[L-2] Lock time isn't factored into rewards

Severity: Low	Status: Acknowledged

Target

- SeraphPool.sol#claim
- SeraphPool.sol#calculateRewardsEarned

Severity:

- Impact: Low
- Likelihood: Medium

Description:

In the staking math implementation, the lock Time Isn't Factored into Rewards

The code has a lock period (minLockPeriod) that prevents unstaking too soon.

However, it doesn't actually boost rewards for longer stakers, it just stops them from unstaking before a certain time.

If the goal was to have time-based multipliers to encourage user stakes for a long period of time, in the math, it's not present here.

If a user stakes right before a reward update and unstakes right after, they might receive a share of that reward without being "long-term."

Recommendation: Consider add time factor to the reward. Users can get more reward if they stakes for a longer period of time.

Bitmind: Acknowledged - Here, we only use the same lock period for stakers for delay purposes and to prevent immediate stake-in->reward->stake-out behavior. Not relating lock time to rewards boost.

[L-3] Admin can remove staking token via `recoverERC20`

Severity: Low	Status: Resolved

Target

- SeraphPool.sol#recoverERC20

Severity:

- Impact: High
- Likelihood: Low

Description: The `recoverERC20` function allows the owner to withdraw any amount of any token, including the staking token. This could drain user deposits if misused.

```
    /**
     * @dev Allows the owner to recover ERC20 tokens mistakenly sent to
the contract.
     * @param _token The address of the ERC20 token to recover.
     * @param _amount The amount of tokens to recover.
     */
    function recoverERC20(address _token, uint256 _amount) external
onlyOwner {
        IERC20(_token).transfer(msg.sender, _amount);
    }
```

Recommendation: Ensure the `recoverERC20` cannot be used to remove the staking token. The function should be only used to remove the reward token.

Bitmind: Addressed with the following commit

Zenith: Verified.

[L-4] Missing min and max lock period validation when setting the _minLockPeriod

Severity: Low	Status: Resolved

Target

- SeraphPool.sol#updateMinLockPeriod

Severity:

- Impact: Low
- Likelihood: Low

Description: The function allows setting any lock period without validation, potentially breaking the staking mechanism with extremely long or zero periods.

Recommendation:

```
function updateMinLockPeriod(uint256 _minLockPeriod) external onlyOwner {
+   if (_minLockPeriod == 0) revert SeraphPool__MinimumLockPeriod();
+   if (_minLockPeriod > 30 days) revert SeraphPool__MaximumLockPeriod();
    minLockPeriod = _minLockPeriod;
}
```

Bitmind: Addressed with the following commit

Zenith: Verified.

[L-5] Unsafe ERC20 Transfer Operations

Severity: Low	Status: Resolved

Target

- SeraphPool.sol#stake
- SeraphPool.sol#unstake

Severity:

- Impact: Medium
- Likelihood: Low

Description: The contract uses `.transfer()` and `.transferFrom()` for ERC20 token transfers without checking return values. Some ERC20 tokens (like USDT) don't follow the standard exactly and may fail silently, leading to potential stuck tokens or failed operations.

Recommendation: Use OpenZeppelin's SafeERC20 library for all token transfers.

```
import { SafeERC20 } from
"@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
using SafeERC20 for IERC20;

// Replace all transfer/transferFrom calls with safe versions:
stakingToken.safeTransfer(msg.sender, _amount);
stakingToken.safeTransferFrom(msg.sender, address(this), _amount);
```

Bitmind: Addressed with this commit

Zenith: Verified.