Seascape Staking
Audit Report

Tue Nov 04 2025

contact@bitslab.xyz https://twitter.com/scalebit_

ScaleBit

Seascape Staking Audit Report

1 Executive Summary

1.1 Project Information

Description	The project implements an LP staking smart contract that provides linear reward distribution and an additional check-in (boost) incentive mechanism.
Type	Staking
Auditors	s3cunda, jolyon
Timeline	Sun Oct 19 2025 - Tue Nov 04 2025
Languages	Solidity
Platform	BSC
Methods	Architecture Review, Unit Testing, Manual Review
Source Code	https://github.com/seascapenetwork/cws-rebrand-smartcontract
Commits	c22142d5050a2e589d08124d7839185c93328aa8 750fa753d51bb7c871e02704af22b0364b765d76 a15ed19495ecbeb5f6048f673520d1a6625bff96 96c8279c706a20480ed081ccb02792385b363a22

1.2 Files in Scope

The following are the SHA1 hashes of the original reviewed files.

ID	File	SHA-1 Hash
BNBNB	src/BNBNB.sol	b2df91f94f2bec161b5509f1d18448ec15a31875
STA1	src/Staking.sol	7e2577cb712cae700b34ce30f93d1ecf48358458
COU	src/Counter.sol	b46140a677c7b27774bbd9039a2436948b8b5443

1.3 Issue Statistic

Item	Count	Fixed	Acknowledged
Total	5	4	1
Critical	0	0	0
Major	2	2	0
Medium	2	2	0
Minor	1	0	1
Informational	0	0	0

1.4 ScaleBit Audit Breakdown

ScaleBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Integer overflow/underflow
- Number of rounding errors
- Unchecked External Call
- Unchecked CALL Return Values
- Functionality Checks
- Reentrancy
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic issues
- Gas usage
- Fallback function usage
- tx.origin authentication
- Replay attacks
- Coding style issues

1.5 Methodology

The security team adopted the **"Testing and Automated Analysis"**, **"Code Review"** and **"Formal Verification"** strategy to perform a complete security test on the code in a way that is closest to the real attack. The main entrance and scope of security testing are stated in the conventions in the "Audit Objective", which can expand to contexts beyond the scope according to the actual testing needs. The main types of this security audit include:

(1) Testing and Automated Analysis

Items to check: state consistency / failure rollback / unit testing / value overflows / parameter verification / unhandled errors / boundary checking / coding specifications.

(2) Code Review

The code scope is illustrated in section 1.2.

(3) Audit Process

- Carry out relevant security tests on the testnet or the mainnet;
- If there are any questions during the audit process, communicate with the code owner in time. The code owners should actively cooperate (this might include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- The necessary information during the audit process will be well documented for both the audit team and the code owner in a timely manner.

2 Summary

This report has been commissioned by Seascape to identify any potential issues and vulnerabilities in the source code of the Seascape Staking smart contract, as well as any contract dependencies that were not part of an officially recognized library. In this audit, we have utilized various techniques, including manual code review and static analysis, to identify potential vulnerabilities and security issues.

During the audit, we identified 5 issues of varying severity, listed below.

ID	Title	Severity	Status
STA-1	Lock of user stake reward	Major	Fixed
STA-2	Theft of reward by "flash-loan" type of attack	Major	Fixed
STA-3	Logical Inconsistency in Boost-Related Conditions	Medium	Fixed
STA-4	Potential out of gas risk in `_checkTimeOverlap` due to unbounded iteration	Medium	Fixed
STA-5	Missing Event Emission in `setGamma` Function	Minor	Acknowledged

3 Participant Process

Here are the relevant actors with their respective abilities within the Seascape Staking Smart Contract :

Owner

- `createSession` : Create a new staking session and fund reward pools.
- `setGamma` : Adjust the gamma parameter before session starts.

User

- `deposit` : Stake tokens into a session.
- `checkIn` : Increase boost points with a 5-minute cooldown.
- `withdraw` : Claim staked tokens and all rewards after session ends.

4 Findings

STA-1 Lock of user stake reward

Severity: Major

Status: Fixed

Code Location:

src/Staking.sol

Descriptions:

In the staking reward calculation logic, rewards are computed based on three factors: total rewards, staking duration, and staking amount. Specifically, the implementation is as follows:

```
uint256 accRewardPerShare = session.accRewardPerShare;

if (block.timestamp > session.lastRewardTime && session.totalStaked > 0) {
    uint256 timeElapsed = _getElapsedTime(_sessionId, session.lastRewardTime);
    uint256 reward = timeElapsed * session.rewardPerSecond;
    accRewardPerShare += (reward * SCALER) / session.totalStaked;
}

stakingReward = (user.amount * accRewardPerShare / SCALER) - user.rewardDebt;
```

The rewardDebt variable, as indicated by its name and code comments, represents the user's cumulative staking rewards. This value is updated when the user stakes (i.e., when calling the deposit function):

```
user.amount += _amount;
user.rewardDebt = user.amount * session.accRewardPerShare / SCALER;
```

When a user decides to end their staking and withdraw both rewards and principal, the reward calculation logic is as follows:

```
uint256 lpReward = (user.amount * session.accRewardPerShare / SCALER) -
user.rewardDebt;
```

From the above code, it is evident that during reward calculation, the user's rewardDebt is deliberately deducted from the total rewards, meaning this portion of rewards is not distributed to the user. Furthermore, no other functions in the code provide an interface for users to withdraw this rewardDebt.

To illustrate this issue clearly, consider the following scenario:

For simplicity in description and calculation, assume the session duration is 100 units, total rewards are 1000 tokens, and the initial totalStaked is 100 tokens. The initial staker is Bob.

1. Alice enters the staking at timeElapsed = 50 with 100 tokens. At this point:

- reward = 500
- accRewardPerShare = 500 / 100 = 5
- After calling processDeposit: amount = 100, rewardDebt = 100 * 5 = 500 totalStaked updates to 200.

2. Alice stakes an additional 100 tokens at timeElapsed = 80:

- reward = 800
- accRewardPerShare = 800 / 200 = 4
- After calling processDeposit: amount = 200, rewardDebt = 200 * 4 = 800 totalStaked updates to 300.

3. When the staking ends at timeElapsed = 100, Alice withdraws her staked funds and rewards:

- reward = 1000
- accRewardPerShare = 1000 / 300 ≈ 3.33
- Alice's lpReward = (200 * 3.33) - 800 ≈ 200 Alice withdraws 200 tokens as rewards, leaving 800 tokens in the contract.

4. Bob ends his staking after Alice:

- reward = 1000
- accRewardPerShare = 1000 / 300 ≈ 3.33
- Bob's lpReward = (100 * 3.33) - 0 ≈ 333 Bob withdraws 333 tokens, leaving 800 - 333 = 467 tokens permanently locked in the contract with no means of recovery.

Suggestion:

It is recommended to refactor the reward distribution section of the code to clarify the purpose of rewardDebt and prevent users' funds from being locked in the contract.

Resolution:

The team adopted our advice and fixed this issue by adding a new variable `accumulatedReward` to record user reawrd and distribute it at the end of staking process.

STA-2 Theft of reward by "flash-loan" type of attack

Severity: Major

Status: Fixed

Code Location:

src/Staking.sol

Descriptions:

In the current contract logic, the distribution of user rewards is solely determined by the amount of staked funds, as shown in the following code snippet:

```
uint256 lpReward = (user.amount * session.accRewardPerShare / SCALER) -
user.rewardDebt;
```

This introduces an issue where the staking duration is completely disregarded, allowing users to stake a large amount of funds shortly before the staking period ends. As a result, such users can easily dilute the reward shares of other participants.

Additionally, while the main reward mechanism currently suffers from another issue (STA01) that prevents proper distribution according to the intended logic, the boost reward is still affected by this type of manipulation, leading to an uneven allocation:

```
  function _calculateCheckInReward(uint256 _sessionId, UserInfo storage user) internal
view returns (uint256) {
      Session storage session = sessions[_sessionId];

      //                 ,     0
      if (user.amount == 0 || session.totalStaked == 0) {
          return 0;
      }

      //                           boost
      if (user.boost == 0) {
          return 0;
```

```
        }

        uint256 totalBoostPool = session.checkInRewardPool;
        uint256 gamma = session.gamma;

        // 1)    stake    : γ * totalBoostPool * (userStaked / totalStaked)
        uint256 stakePart = (totalBoostPool * gamma) / SCALER;
        uint256 stakeShare1e18 = (user.amount * SCALER) / session.totalStaked;
        uint256 rewardFromStake = (stakePart * stakeShare1e18) / SCALER;

        // 2)    hybrid    : (1-γ) * totalBoostPool * (s_i*b_i) / Σ(s*b)
        uint256 hybridPart = totalBoostPool - stakePart;
        uint256 rewardFromHybrid = 0;

        //      :         (totalBoostPoints==0),   hybrid      stake
        if (session.totalBoostPoints == 0) {
            rewardFromHybrid = (hybridPart * stakeShare1e18) / SCALER;
        } else {
            //        s_i * b_i
            // s_i = user.amount / totalStaked (scaled by 1e18)
            // b_i = user.boost / totalBoostPoints (scaled by 1e18)
            // s_i * b_i = (s_i * b_i) / 1e18
            uint256 b1e18 = (user.boost * SCALER) / session.totalBoostPoints;
            uint256 sb1e18 = (stakeShare1e18 * b1e18) / SCALER;

            // Σ(s*b)      :
            //    Σ(s_i * b_i) = Σ((stake_i/totalStaked) * (boost_i/totalBoost))
            //             = (1/(totalStaked * totalBoost)) * Σ(stake_i * boost_i)
            //             = totalWeightedStake / (totalStaked * totalBoost)
            //    sb_i      1e18 scaled, sumSB       1e18 scaled
            // sumSB1e18 = (totalWeightedStake / totalStaked) * (SCALER / totalBoostPoints)
            //        = (totalWeightedStake * SCALER) / (totalStaked * totalBoostPoints)
            uint256 sumSB1e18 = (session.totalWeightedStake * SCALER) /
(session.totalStaked * session.totalBoostPoints);

            if (sumSB1e18 > 0) {
                rewardFromHybrid = (hybridPart * sb1e18) / sumSB1e18;
            } else {
                // fallback:   stake
                rewardFromHybrid = (hybridPart * stakeShare1e18) / SCALER;
            }
        }

        return rewardFromStake + rewardFromHybrid;
    }
```

Suggestion:

It is recommended to refactor the reward distribution function, as the current logic is overly convoluted. It is advisable to take both staking duration and staked amount into account when distributing rewards.

Resolution:

The team adopted our advice and fixed this issue by adding a new variable `accumulatedReward` to record user reawrd, which take both staking duration and staked amount into account when distributing rewards.

STA-3 Logical Inconsistency in Boost-Related Conditions

Severity: Medium

Status: Fixed

Code Location:

src/Staking.sol#475

Descriptions:

In the `_calculateCheckInReward` function, the conditions `user.boost == 0` and `session.totalBoostPoints == 0` introduce logical inconsistency.

```
if (user.boost == 0) {
    return 0;
}
```

This means any user with boost == 0 will directly return 0 and exit the function.

However, in the later logic:

```
if (session.totalBoostPoints == 0) {
    rewardFromHybrid = (hybridPart * stakeShare1e18) / SCALER;
}
```

the code assumes that totalBoostPoints can be 0, and in that case, all users will receive hybrid rewards based purely on stake share.

Suggestion:

Clarify the intended design and refactor the logic accordingly:

If boost == 0 users should still get hybrid rewards, remove the early return check.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

STA-4 Potential out of gas risk in `_checkTimeOverlap` due to unbounded iteration

Severity: Medium

Status: Fixed

Code Location:

src/Staking.sol#615

Descriptions:

The `_checkTimeOverlap` function iterates over the entire sessionTimeRanges array to check if a new session overlaps with any existing sessions:

```
function _checkTimeOverlap(uint256 _startTime, uint256 _endTime) internal view {
    for (uint256 i = 0; i < sessionTimeRanges.length; i++) {
        TimeRange memory range = sessionTimeRanges[i];

        //           :
        bool overlap = (_startTime < range.endTime && _endTime > range.startTime);

        require(!overlap, "Session time overlaps with existing session");
    }
}
```

If the array grows too large, this iteration could cause out-of-gas errors when creating a new session.

Suggestion:

Ensure that sessions are created in strictly chronological order (each new session starts after the previous session ends). With this guarantee, it is sufficient to check only that _startTime is greater than the endTime of the last session.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

STA-5 Missing Event Emission in `setGamma` Function

Severity: Minor

Status: Acknowledged

Code Location:

src/Staking.sol#216

Descriptions:

```
function setGamma(uint256 _sessionId, uint256 _gamma)
    external
    onlyOwner
    sessionExists(_sessionId)
{
    Session storage session = sessions[_sessionId];
    require(block.timestamp < session.startTime, "Cannot modify gamma after session started");
    require(_gamma <= SCALER, "Gamma must be <= 1e18");

    session.gamma = _gamma;
}
```

The `setGamma` function updates the gamma parameter for a given session but does not emit any event after the change.

Suggestion:

Emit an event after successfully updating the gamma.

Appendix 1

Issue Level

- **Informational** issues are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information at risk. All critical issues should be fixed.

Issue Status

- **Fixed:** The issue has been resolved.
- **Partially Fixed:** The issue has been partially resolved.
- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

Appendix 2

Disclaimer