Stable Jack Audit Report

Version 1.0

Conducted by: Kiki

April 2025

Table of Contents

1 Introduction

1.1 About Kiki

Kiki is a Independent Security Researcher who has conducted dozens of security reviews with the top security firm Guardian Audits, as well as through private engagements. View their previous work here, or reach out via Twitter or Telegram.

1.2 Disclaimer

Security Reviews are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

1.3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

1.3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

1.3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

2 Executive Summary

2.1 Overview

Project Name	Stable Jack
Codebase	jack-tokenomics
Operating platform	Avalanche
Language	Solidity
Initial commit	[1] 6e5838c76782ef963a396a571e6bf94187cd2d69
Remediation commit	8abf9b15fd78f0c4238b166b2f2328bbb1d83d00
Audit methodology	Manual Review

2.2 Scope

Files and folders in scope

- src/Staking
- src/Treasury.sol
- src/JackTokenBridged.sol
- src/SeedLaunchVesting.sol
- src/Vesting.sol
- src/AirDropStaking.sol
- src/JackToken.sol

2.3 Issues Found

Severity	Total Found	Resolved	Partially Resolved	Acknowledged
Critical risk	2	2	0	0
High risk	5	5	0	0
Medium risk	7	4	0	3
Low risk	14	9	1	4

2.4 Findings & Resolutions

ID	Title	Severity	Status
C-01	Share Transfers Enable Unlimited Reward Generation Attack	Critical	Resolved
C-02	Phantom Rewards Due to Missing remainingRewards Update	Critical	Resolved
H-01	Inaccurate Reward Accounting Enables Overstaking	High	Resolved
H-02	Withdrawal & Redeem DoS via Third-Party Deposits	High	Resolved
H-03	Missing Claimed Rewards in unusedTokens Calculation	High	Resolved
H-04	Decreasing Reward Rate Due to Incorrect Metric	High	Resolved
H-05	Admin Recovery Function Will Steal User Funds Across All Schedules	High	Resolved
M-01	Incorrect Exit Fee Deduction For unusedTokens	Medium	Resolved
M-02	Persistent Stake Agreements Lead to Permanent Staking Lockout	Medium	Resolved
M-03	TGE cliff Omission in Final Vesting Claims	Medium	Acknowledged
M-04	Integer truncation enables token discount exploit	Medium	Acknowledged
M-05	Unbounded purchases array enables gas DoS	Medium	Acknowledged
M-06	Stale Rewards Calculation in totalAssets	Medium	Resolved
M-07	Missing Native Currency Recovery Mechanism	Medium	Resolved
L-01	No Minimum Stake Amount Enables Dust Attacks	Low	Resolved
L-02	Vesting Schedule Expiration Check Misses Boundary	Low	Acknowledged
L-03	Missing totalStaked Update After Claim	Low	Resolved
L-04	Inefficient Duplicate Code in claim Function	Low	Resolved
L-05	Incorrect maxRedeem Calculation Can Cause Reverts	Low	Resolved
L-06	Missing Slippage Control	Low	Resolved
L-07	Incorrect Operation Order in mint Function	Low	Resolved
L-08	Circumvention of minimum requirement	Low	Resolved
L-09	Unchecked Minimum Deposit Limit Update	Low	Resolved
L-10	Unclear Reward Routing on ERC4626 Redemption	Low	Acknowledged
L-11	Admin Can End Sale Before Advertised End Time	Low	Acknowledged
L-12	Precision Loss in Vesting Calculations	Low	Partially Resolved

ID	Title	Severity	Status
L-13	Inconsistent Rounding in Vesting Calculations	Low	Resolved
L-14	Centralization Risk	Low	Acknowledged

3 Findings

3.1 Critical Risk

3.1.1 Share Transfers Enable Unlimited Reward Generation Attack

Severity: *Critical risk (Resolved)*

Context: ***AirDropStaking.sol***:*436*

Description:

The `AirDropStaking` contract possesses a vulnerability in its reward accounting mechanism when shares are transferred between users. This vulnerability stems from how the `earned` function calculates rewards without tracking share ownership changes:

```
function earned(address account) public view returns (uint256) {
    return
        FullMath.mulDiv(
            balanceOf(account),
            rewardPerShareStored - userRewardPerSharePaid[account],
            REWARD_PRECISION
        ) + rewards[account];
}
```

The reward calculation system works as follows:

1. The contract maintains a global `rewardPerShareStored` value that increases as rewards accumulate
2. Each user has a `userRewardPerSharePaid` value tracking the point at which they last claimed rewards
3. A user's earned rewards are calculated as: `currentShares * (currentGlobalRate - userLastClaimedRate)`
4. When users deposit or claim, their `userRewardPerSharePaid` is updated to the current `rewardPerShareStored`

However, when shares are transferred via ERC20 transfer functions, the `userRewardPerSharePaid` value is not updated for either sender or receiver. This creates a severe accounting flaw:

- A new user receiving transferred shares will have a default `userRewardPerSharePaid` of 0
- The calculation `rewardPerShareStored - userRewardPerSharePaid[newUser]` will return the entire historical accumulation of rewards per share
- Multiplied by the transferred share balance, this grants the new user rewards for the entire history of those shares

This can be exploited through a systematic attack:

1. Attacker deposits tokens and receives shares
2. Attacker transfers shares to a fresh wallet with no reward history
3. The fresh wallet claims excessive rewards based on the full share history

4. When the fresh wallet calls `redeem()` or `claimRewards()`, it receives both the original deposit and fabricated rewards
5. This process can be repeated with multiple wallets

If executed at scale, this attack will drain:

1. First, all reward tokens allocated for the current epoch
2. Eventually, all staked JACK tokens in the vault when the `totalWithdrawal = assets + reward` calculation in `redeem()` combines the base assets with artificial rewards

This attack allows for unlimited artificial reward generation that can eventually drain all funds from the contract.

Recommendation:

Disable share transfers entirely by overriding the ERC20 transfer functions to revert:

```
function transfer(address to, uint256 amount) public override returns (bool) {
    revert("Share transfers not permitted");
}

function transferFrom(address from, address to, uint256 amount) public override
    returns (bool) {
    revert("Share transfers not permitted");
}
```

This approach ensures that rewards remain tied to the original depositor and prevents the creation of artificial rewards through share transfers. Since the contract is designed as a staking vault rather than a freely transferable asset, this limitation should align with its intended use case.

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.1.2 Phantom Rewards Due to Missing remainingRewards Update

Severity: *Critical risk (Resolved)*

Context: ***AirDropStaking.sol***:405-426

Description:

The `_calculateRewardPerShare` function in the `AirDropStaking` contract calculates rewards to distribute but never reduces `currentEpoch.remainingRewards` to reflect these distributions:

```
uint256 rewardsToDistribute = (currentEpoch.remainingRewards *
    timeElapsed) / epochDuration;
```

After calculating and distributing rewards through the reward-per-share mechanism, the `remainingRewards` state variable should be decreased, but this critical update is missing. This creates a severe accounting error that cascades through multiple contract functions:

1. The `addRewards` function will fail when creating a new epoch, as it calculates `totalRewardForNextEpoch = currentEpoch.remainingRewards + rewardAmount`. Since `remainingRewards` never decreases, this sum will be inflated, causing the contract to believe more tokens are needed than actually required.

2. The `totalAssets` function calculates available assets as `jackToken.balanceOf(address(this))- currentEpoch.remainingRewards`. With an artificially high `remainingRewards` value, this function will increasingly underreport the vault's assets, eventually leading to an arithmetic underflow when `remainingRewards` exceeds the contract balance.
3. Since core ERC4626 functions (`deposit`, `withdraw`, `mint`, `redeem`) depend on `totalAssets`, the entire vault functionality will eventually become unusable when the underflow occurs.

This bug effectively creates "phantom rewards" that exist in accounting but aren't actually distributed, causing progressively worsening calculation errors over time.

Recommendation:

Update `currentEpoch.remainingRewards` by subtracting `rewardsToDistribute`:

```
// Calculate rewardsToDistribute

// Ensure we don't distribute more than what's available
if (rewardsToDistribute > currentEpoch.remainingRewards) {
    rewardsToDistribute = currentEpoch.remainingRewards;
}
currentEpoch.remainingRewards -= rewardsToDistribute;

// Rest of the function...
```

This will ensure that if any rewards are remaining when `addRewards` are called they are accounted for but not inflated.

Additionally introduce a new variable such as `currentEpoch.remainingRewardsInContract`. The purpose of this variable will be to accurately track `totalAssets`. When `addRewards` is called this should be increased by `rewardAmount` and then every time rewards actually leave the contract via `claimRewards` or `redeem`. The `currentEpoch.remainingRewardsInContract` should be reduced by that amount. This way the `totalAssets` function accurately reduces the balance of the contract by exactly the amount of rewards remaining in the contract.

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.2 High Risk

3.2.1 Inaccurate Reward Accounting Enables Overstaking

Severity: *High risk (Resolved)*

Context: ***Staking.sol***:*250-253*

Description:

The `Staking` contract incorrectly calculates whether a new stake would exceed the allocated tokens. The calculation only considers `schedule.maxPotentialRewards + potentialReward` without including rewards that have already been claimed (`schedule.claimedRewards`).

This arithmetic error creates a vulnerability in the reward accounting system. As users claim rewards, the `schedule.maxPotentialRewards` value decreases, but the validation doesn't account for these

already-claimed rewards. This permits additional stakes to be created beyond what the allocated tokens can support.

For example, consider this scenario:

- `schedule.allocatedTokens = 100e18`
- `schedule.claimedRewards = 70e18` (70% of rewards already claimed)
- `schedule.maxPotentialRewards = 0` (no current outstanding stakes)

When a user tries to stake an amount that would yield 100e18 in rewards (the full allocated tokens amount), the following validation occurs:

```
if (schedule.allocatedTokens < schedule.maxPotentialRewards + potentialReward)
    IStaking.InsufficientAllocation.selector.revertWith();

// With our scenario values:
if (100e18 < 0 + 100e18)  // This doesn't revert
```

Despite 70% of the rewards already being claimed, the user is able to stake an amount that will yield them 100% of the initially allocated rewards. This happens because when users claim or exit early, `schedule.maxPotentialRewards` is decreased:

```
schedule.maxPotentialRewards -= potentialReward;
```

However, `schedule.allocatedTokens` remains unchanged, and `schedule.claimedRewards` (which is updated during claims) is never used in the stake validation.

The consequences are severe: later stakers may be unable to claim their promised rewards because the contract has already distributed more rewards than anticipated. This could lead to:

1. Contract insolvency for reward distributions
2. Some stakers being unable to receive their rewards & initial stake

Recommendation:

Modify the calculation to account for claimed rewards by using:

```
if (schedule.maxPotentialRewards + schedule.claimedRewards + potentialReward >
    schedule.allocatedTokens) {
    IStaking.InsufficientAllocation.selector.revertWith();
}
```

This ensures the sum of outstanding potential rewards, claimed rewards, and new potential rewards cannot exceed the allocated tokens, maintaining proper accounting integrity.

Resolution: Resolved. The recommended fix was implemented in **9db0ad0**.

3.2.2 Withdrawal & Redeem DoS via Third-Party Deposits

Severity: *High risk (Resolved)*

Context: ***AirDropStaking.sol***:212-213

Description:

The `deposit` function in `AirDropStaking` allows any user to deposit tokens on behalf of another user, which updates the recipient's `timestamp` to `block.timestamp`. This timestamp is used to enforce a cooling period before users can withdraw their funds.

An attacker can exploit this design by repeatedly making small deposits (at the minimum deposit amount) to a victim's account. Each deposit refreshes the cooling period timestamp, effectively preventing the victim from withdrawing their funds indefinitely, as long as the attacker is willing to continue making the minimum deposits required by the contract. While this will come at a cost to the attacker and the victim is gaining funds from such an attack, they will not have access to these funds nor the yield from staking.

Recommendation:

Modify the `deposit` function to restrict depositing to only the `msg.sender`.

Resolution: Resolved. The recommended fix was implemented in **9db0ad0**.

3.2.3 Missing Claimed Rewards in unusedTokens Calculation

Severity: *High risk (Resolved)*

Context: ***Staking.sol***:353

Description:

The `endSchedule` function in the `Staking` contract contains an accounting error when calculating the amount of unused tokens to return to the treasury:

```
uint256 unusedTokens = schedule.allocatedTokens -
    schedule.maxPotentialRewards -
    schedule.earlyExitFees;
```

This calculation fails to consider tokens that have already been claimed by stakers (`schedule.claimedRewards`). When rewards are claimed, `schedule.claimedRewards` increases, and by not including this value in the calculation, the contract miscalculates the actual remaining unused tokens.

This error leads to insufficient funds for remaining stakers. When `schedule.claimedRewards` is significant, the formula returns a larger `unusedTokens` value than actually available. After transferring this inflated amount to the treasury, the contract will have insufficient funds to pay the remaining stakers their promised rewards.

For example, with:

- `schedule.allocatedTokens = 100 tokens`
- `schedule.maxPotentialRewards = 60 tokens` (rewards owed to active stakers)
- `schedule.claimedRewards = 40 tokens` (already paid out)
- `schedule.earlyExitFees = 0`
- Contract reward balance = 60 tokens
- Amount to be transferred = 40 (100 - 60 - 0 = 40)
- Contract reward balance after transfer = 20 tokens

The current calculation would transfer 40 tokens to the treasury, leaving only 20 tokens to cover 60 tokens of obligations to stakers (`maxPotentialRewards`).

Recommendation:

Modify the `unusedTokens` calculation to include `schedule.claimedRewards`. This ensures that only genuinely unused tokens are returned to the treasury, preserving funds needed for remaining stakers and preventing transaction failures due to insufficient balances.

Resolution: Resolved. The recommended fix was implemented in **9db0ad0**.

3.2.4 Decreasing Reward Rate Due to Incorrect Metric

Severity: *High risk (Resolved)*

Context: ***AirDropStaking.sol**:415*

Description:

In the `_calculateRewardPerShare` function of the `AirDropStaking` contract, rewards distribution calculations incorrectly use `currentEpoch.remainingRewards` instead of `currentEpoch.totalRewards`:

```
uint256 rewardsToDistribute = (currentEpoch.remainingRewards *
    timeElapsed) / epochDuration;
```

This creates a decreasing reward rate, as `remainingRewards` diminishes over the epoch duration. When calculating how many rewards to distribute for a given time interval, the contract should use the total rewards allocated for the entire epoch to ensure consistent distribution.

With the current implementation, early interactions with the contract receive disproportionately higher rewards than later ones. For example:

- If `totalRewards` is 1000 tokens for a 10-day period:
  - Day 1: 10% of 1000 = 100 tokens distributed
  - `remainingRewards` reduces to 900
  - Day 2: 10% of 900 = 90 tokens distributed (instead of 100)
  - `remainingRewards` reduces to 810
  - Day 3: 10% of 810 = 81 tokens distributed (instead of 100)

By the end of the epoch, a significant portion of rewards remains undistributed, violating the expected behavior of distributing all rewards evenly over the epoch duration.

This error creates an unfair advantage for users who stake early in the epoch and penalizes those who stake later, contradicting the linear distribution design.

It is important to note that this issue will not present itself because of another unrelated issue with `remainingRewards`; however, with the protocol working as expected, this issue will present itself and severely diminish users' yield.

Recommendation:

Modify the reward calculation to use `totalRewards` instead of `remainingRewards`:

```
uint256 rewardsToDistribute = (currentEpoch.totalRewards *
    timeElapsed) / epochDuration;
```

This ensures that rewards are distributed at a constant rate throughout the entire epoch, regardless of when users interact with the contract.

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.2.5 Admin Recovery Function Will Steal User Funds Across All Schedules

Severity: *High risk (Resolved)*

Context: ***Staking.sol***:*367-384*

Description:

The `recoverFromEndedSchedule` function in the `Staking` contract has a vulnerability that allows admins to transfer tokens that belong to users from not only the targeted schedule but potentially all schedules in the contract:

```
function recoverFromEndedSchedule(uint256 scheduleId) external onlyRole(
    DEFAULT_ADMIN_ROLE) {
    StakeSchedule storage schedule = schedules[scheduleId];
    if (schedule.status != ScheduleStatus.Ended)
        IStaking.ScheduleNotEnded.selector.revertWith();
    if (block.timestamp <= schedule.endTime + RECOVERY_DELAY)
        IStaking.TooEarlyToRecover.selector.revertWith();

    uint256 expectedTokens = schedule.pendingRewards + schedule.earlyExitFees;
    uint256 currentBalance = stakingToken.balanceOf(address(this));
    if (currentBalance > expectedTokens) {
        uint256 recoverable = currentBalance - expectedTokens;
        stakingToken.safeTransfer(address(treasury), recoverable);
        emit TreasuryRecovery(scheduleId, recoverable);
    }
}
```

The flaw is twofold:

1. The function uses `stakingToken.balanceOf(address(this))` which returns the total balance of the contract across all schedules, not just the targeted schedule. This means the calculation is comparing schedule-specific accounting with global token balance.
2. The function only subtracts `schedule.pendingRewards` and `schedule.earlyExitFees` from the total balance to determine what's "recoverable." This ignores:
    - Unclaimed user stake principal in the targeted schedule
    - All user stakes and rewards from other active schedules
    - Pending rewards from other schedules

When this function is called, it can potentially transfer out tokens that represent:

- User principal amounts from the targeted schedule

- User principal amounts from other active schedules
- Pending rewards from other schedules

This could result in fund loss for users across the entire contract, with users unable to claim their stakes and rewards because the tokens have been transferred to the treasury.

Recommendation:

Redesign the recovery system with two separate functions:

1. A force claim function that allows admins to process claims on behalf of users who haven't claimed after the schedule has ended:

```
function forceClaimForUser(uint256 scheduleId, address user, uint256 agreementId)
    external onlyRole(DEFAULT_ADMIN_ROLE) {
    // Process claim for user and send them their stake + rewards or just stake
        minus penalty
}
```

1. A schedule-specific recovery function that only recovers genuinely unused rewards:

```
function recoverUnusedRewards(uint256 scheduleId) external onlyRole(
    DEFAULT_ADMIN_ROLE) {
    // Only recover tokens that are truly not owed to anyone in this specific
        schedule
    uint256 unusedRewards = schedule.allocatedTokens - schedule.claimedRewards -
        schedule.maxPotentialRewards;
    // Transfer only unusedRewards to treasury
}
```

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.3 Medium Risk

3.3.1 Incorrect Exit Fee Deduction For unusedTokens

Severity: *Medium risk (Resolved)*

Context: ***Staking.sol***:355

Description:

The `endSchedule` function in the `Staking` contract incorrectly includes `schedule.earlyExitFees` when calculating the `unusedTokens` to be returned to the treasury:

```
uint256 unusedTokens = schedule.allocatedTokens -
    schedule.maxPotentialRewards -
    schedule.earlyExitFees;
```

This is a conceptual accounting error because `unusedTokens` should represent only unused reward tokens from the original allocation. The foundation of `unusedTokens` is based on `schedule.allocatedTokens`, which is never impacted by `schedule.earlyExitFees` changes.

The `schedule.earlyExitFees` represents penalties collected from users who exited their stakes early, which is fundamentally different from the reward token accounting. Including `schedule.earlyExitFees` in this calculation artificially reduces the amount of unused tokens returned to the treasury.

Recommendation:

Remove `schedule.earlyExitFees` from the `unusedTokens` calculation.

Resolution: Resolved. The recommended fix was implemented in **9db0ad0**.

3.3.2 Persistent Stake Agreements Lead to Permanent Staking Lockout

Severity: *Medium risk (Resolved)*

Context: ***Staking.sol**:300*

Description:

In the `stake` function of the `Staking` contract, stake agreements are permanently stored in the contract's state even after they've been claimed. Instead, they are marked as claimed, preventing them from being maliciously reused.

```
function claim(uint256 scheduleId, uint256 agreementId) external ... {
    // ...
    agreement.claimed = true;
    // Agreement remains in storage
    // ...
}
```

Since the contract enforces `MAX_AGREEMENTS_PER_USER = 50`, users will eventually be unable to create new stakes once they reach this limit, even if they've claimed all their previous stakes. This creates a permanent denial of service condition where users who have been actively staking and claiming must use a new address to continue staking.

This issue is particularly impactful for long-term or frequent stakers who might reasonably execute more than 50 stakes during their participation in a specific schedule.

Recommendation:

Consider modifying the `claim` function to remove agreements from storage after they've been claimed since the agreement is no longer usable.

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.3.3 TGE cliff Omission in Final Vesting Claims

Severity: *Medium risk (Acknowledged)*

Context: ***Vesting.sol**:231*

Description:

In the `startSchedule` function, the end time validation fails to account for the TGE cliff period in its calculation:

```
if (_endTime != (_startTime + _installments * _installmentInterval)) {
    IVesting.ErrorEndTimeMismatch.selector.revertWith();
}
```

This creates a timing mismatch because:

1. The schedule end time is set to: `startTime + (installments * interval)`
2. But the actual time needed for all installments is: `startTime + tgeCliff + (installments * interval)`
3. The `getTotalUnlocked` function calculates vesting based on time since TGE cliff:

```
uint256 timePassed = block.timestamp - (s.startTime + s.tgeCliff);
uint256 intervalsPassed = timePassed / s.installmentInterval;
```

As a result, when the final installment becomes unlockable (after TGE cliff + all intervals), the Treasury schedule will have already expired since it uses the shorter duration.

It is important to note that the `tgeCliff` is also missing in the `MAX_VESTING_DURATION` check of the `SeedLaunchVesting` contract. Where similarly only the product of `_installments` and `_installmentInterval` are taken into consideration.

Recommendation:

Modify the end time validation in `startSchedule` to include the TGE cliff period.

Resolution: Acknowledged by the team.

3.3.4 Integer truncation enables token discount exploit

Severity: *Medium risk (Acknowledged)*

Context: ***SeedLaunchVesting.sol***:287-288

Description:

In the `buyTokens` function of the `SeedLaunchVesting` contract, there is a flaw in the token pricing calculation that allows users to receive more tokens than they should for the payment made:

```
// Calculate and process payment
uint256 cost = (_amountToBuy * s.pricePerToken) / TOKEN_DECIMALS_FACTOR;
s.paymentToken.safeTransferFrom(msg.sender, s.paymentCollector, cost);

// Record purchase
purchases[_scheduleId][msg.sender].push(
    Purchase({
        amount: _amountToBuy,
        purchaseTime: block.timestamp,
        claimed: 0
    })
);
```

This implementation creates a pricing asymmetry due to integer division truncation. When calculating `cost`, any remainder from the division is discarded, effectively rounding down the payment amount. However, the contract still credits users with their full requested `_amountToBuy` tokens.

The issue is particularly severe with small token purchases and/or smaller decimal tokens as the `paymentToken`, where the truncation effect is proportionally larger.

Even with larger purchases, users will consistently receive more tokens than they should have based on the paid amount.

This vulnerability becomes more impactful when:

1. Token prices are set at values that frequently cause truncation
2. The token has high value, magnifying the economic impact
3. Users exploit the issue by making multiple small purchases

Recommendation:

Modify the token purchase calculation to ensure truncation does not lead to a purchasing discount by recalculating the actual amount of tokens a user should receive based on the computed cost:

```
// First calculate cost based on requested amount
uint256 cost = (_amountToBuy * s.pricePerToken) / TOKEN_DECIMALS_FACTOR;
// Then recalculate exact amount user should receive based on cost
_amountToBuy = (cost * TOKEN_DECIMALS_FACTOR) / s.pricePerToken;

// Continue with validations using adjusted _amountToBuy
if (_amountToBuy == 0) {
    ISeedLaunchVesting.ErrorZeroPurchase.selector.revertWith();
}
```

Resolution: Acknowledged by the team.

3.3.5 Unbounded purchases array enables gas DoS

Severity: *Medium risk (Acknowledged)*

Context: ***SeedLaunchVesting.sol***:291

Description:

The `SeedLaunchVesting` contract has a design issue where the `purchases` array for a user can grow without any upper bound:

```
purchases[_scheduleId][msg.sender].push(
    Purchase({
        amount: _amountToBuy,
        purchaseTime: block.timestamp,
        claimed: 0
    })
);
```

Each time a user calls the `buyTokens` function, a new entry is added to their purchases array without any check on its maximum length. This unbounded growth creates a risk of self-denial-of-service in the `claim` function that iterates through this array:

```
for (uint256 i = 0; i < userPs.length; i++) {
    Purchase storage p = userPs[i];
    uint256 unlocked = _getUnlockedAmount(s, p);
    // ...
}
```

Given that a single schedule could be active for a long period of time it is likely that a user could make many purchases and they could inadvertently create a situation where this function consume more gas than the block limit allows, preventing them from claiming their tokens. This is primarily a self-DoS vulnerability since it only affects the user who created the excessive array entries.

Recommendation:

Consider adding a reasonable maximum limit to the number of purchases a user can have for a given schedule.

Resolution: Acknowledged by the team.

3.3.6 Stale Rewards Calculation in totalAssets

Severity: *Medium risk (Resolved)*

Context: ***AirDropStaking.sol***:467

Description:

The `totalAssets` function incorrectly calculates the total assets in the vault by using a potentially stale value for `currentEpoch.remainingRewards`:

```
function totalAssets() public view override(ERC4626, IERC4626) returns (uint256) {
    return jackToken.balanceOf(address(this)) - currentEpoch.remainingRewards;
}
```

This calculation fails to account for pending reward accruals between reward update events. Between these update events, rewards continue to accrue to stakers based on time elapsed but are in a "pending" state until an update occurs, at which point `rewardPerShareStored` is updated. This creates a timing inconsistency:

1. At time t0, `_updateReward` is called and `currentEpoch.remainingRewards` is properly set.
2. Between t0 and t1, additional rewards are earned by stakers based on elapsed time.
3. At t1, if `totalAssets` is called before any new transaction triggers `_updateReward`, it will use the outdated value from t0.

This results in the `totalAssets` function undervaluing the vault's true asset value between reward updates by subtracting more rewards than should actually be excluded, most consistently impacting `getNavPerShare`.

Recommendation:

Modify the `totalAssets` function to calculate up-to-date rewards similar to how `_calculateRewardPerShare` works.

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.3.7 Missing Native Currency Recovery Mechanism

Severity: *Medium risk (Resolved)*

Context: ***JackToken.sol***:22

Description:

The `JackToken` contract implements functionality for cross-chain token transfers via LayerZero, making use of the `OFTCore` which includes the `lzReceive` function. This function is payable, allowing it to receive native currency as part of cross-chain operations. However, the contract lacks a mechanism to recover any native currency that may become trapped in the contract.

While the contract does implement ERC20 token recovery through the `recoverERC20` function, there is no equivalent functionality for native currency. This oversight could result in native currency becoming permanently locked in the contract if it's sent via the `lzReceive` function or if someone directly sends ETH to the contract address through a self-destruct operation. Over time, this could lead to a loss of funds that cannot be recovered.

This applies to the `JackTokenBridged` contract as well.

Recommendation:

Implement a function that allows the contract administrator to recover native currency that might accumulate in the contract.

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.4 Low Risk

3.4.1 No Minimum Stake Amount Enables Dust Attacks

Severity: *Low risk (Resolved)*

Context: ***Staking.sol**:237*

Description:

The `stake` function in the `Staking` contract only validates that the staking amount is not zero without enforcing a minimum stake amount threshold:

```
if (amount == 0) IStaking.InvalidAmount.selector.revertWith();
```

This insufficient validation allows users to create "dust stakes" with extremely small amounts. These dust stakes can lead to several negative consequences:

1. Dust stakes may generate zero rewards due to rounding errors in reward calculations, consuming storage and processing time without providing intended benefits.
2. By distributing funds across many tiny stakes, users can withdraw their funds while not paying penalties, circumventing the intended economic disincentives.
3. Dust stakes can be created even when there are no rewards available due to max staking already being reached.

This validation gap allows users to interact with the staking system in ways that deviate from the intended economic model.

Recommendation:

Implement a minimum stake amount threshold by modifying the validation to check against a defined constant:

```
if (amount < MIN_STAKE_AMOUNT) IStaking.InvalidAmount.selector.revertWith();
```

Define `MIN_STAKE_AMOUNT` as a constant with a value that prevents dust-based manipulation strategies (e.g., `1e6` or something similar).

Resolution: Resolved. The recommended fix was implemented in **9db0ad0**.

3.4.2 Vesting Schedule Expiration Check Misses Boundary

Severity: *Low risk (Acknowledged)*

Context: ***Treasury.sol**:210*

Description:

In the `Treasury` contract, the `pullForVesting` function checks whether a vesting schedule has expired using an incorrect comparison operator:

```
if (
    vestingSchedule.validUntil != 0 &&
    block.timestamp > vestingSchedule.validUntil
) {
    ITreasury.ErrorScheduleExpired.selector.revertWith();
}
```

This validation uses a strict "greater than" (`>`) comparison instead of "greater than or equal to" (`>=`). As a result, if `block.timestamp` is exactly equal to `vestingSchedule.validUntil`, the function would not revert, allowing a vesting contract to pull tokens at the exact moment the schedule has expired.

The `validUntil` timestamp represents the moment when the schedule becomes invalid, meaning any attempt to pull tokens at or after this timestamp should be rejected. The current implementation creates a one-second window where a token pull operation could be processed even though the schedule has technically expired.

This also applies to the check in the `deleteSchedule` function.

Recommendation:

Modify the comparison operator to properly check for expiration by using greater than or equal to (`>=`).

Resolution: Acknowledged by the team.

3.4.3 Missing totalStaked Update After Claim

Severity: *Low risk (Resolved)*

Context: ***Staking.sol**:302*

Description:

The `Staking` contract fails to update the `schedule.totalStaked` state variable when the `claim` function is called. At a point where staked tokens are being processed and returned to users, the code correctly handles the token transfer of the `amount` but omits reducing `schedule.totalStaked` by the corresponding `amount`.

This accounting error causes the contract to maintain an inflated value for `schedule.totalStaked`, which becomes increasingly inaccurate as more tokens are claimed. The `schedule.totalStaked` variable is intended to track the total amount of tokens currently staked in a particular schedule, but due to this omission, it only accurately reflects stakes without accounting for claims.

This accounting issue leads to several consequences:

1. The contract reports incorrect data about the total amount staked.
2. Any functions that rely on `schedule.totalStaked` will operate with incorrect information.
3. Protocol statistics displayed to users will be inaccurate.
4. Any calculations based on the total staked amount (such as percentage-based metrics or TVL) will be wrong.

Recommendation:

Add the missing state update in the `claim` function where staked tokens are being removed:

```
schedule.totalStaked -= amount;
```

This addition ensures accurate accounting of the total staked amount throughout the lifecycle of stakes in the protocol.

Resolution: Resolved. The recommended fix was implemented in **9db0ad0**.

3.4.4 Inefficient Duplicate Code in claim Function

Severity: *Low risk (Resolved)*

Context: ***Staking.sol***:*305-317*

Description:

In the `claim` function of `Staking.sol`, the contract inefficiently reduces `schedule.pendingRewards` by the same `potentialReward` amount in both branches of the if-else statement:

```
if (block.timestamp >= agreement.endTime) {
    schedule.claimedRewards += potentialReward;
    schedule.pendingRewards -= potentialReward;
    schedule.totalRewards += potentialReward;
    // ...
} else {
    uint256 penalty = (amount * schedule.earlyExitPenalty) / BPS;
    schedule.pendingRewards -= potentialReward;
    schedule.earlyExitFees += penalty;
    // ...
}
```

This duplication creates unnecessary code complexity and increases the risk of inconsistency if future updates modify one branch but not the other. It contradicts the pattern used earlier in the same function, where `schedule.maxPotentialRewards` is properly updated once before the conditional logic:

```
agreement.claimed = true;
schedule.maxPotentialRewards -= potentialReward;
```

While this issue doesn't affect functionality, it represents a code quality concern and could lead to maintenance problems over time.

Recommendation:

Move the `schedule.pendingRewards` reduction before the if-else statement to improve code clarity and maintainability:

Resolution: Resolved. The recommended fix was implemented in **9db0ad0**.

3.4.5 Incorrect maxRedeem Calculation Can Cause Reverts

Severity: *Low risk (Resolved)*

Context: ***AirDropStaking.sol***:*185*

Description:

The `AirDropStaking` contract incorrectly overrides the `maxRedeem` function from the ERC4626 standard by adding rewards to the share balance:

```
function maxRedeem(address owner) public view override(ERC4626, IERC4626) returns (
    uint256) {
    return balanceOf(owner) + rewards[owner];
}
```

This implementation fundamentally misunderstands the purpose of `maxRedeem`, which should return the maximum number of shares a user can redeem, not the expected asset amount. Rewards in this contract are not shares—they're additional assets distributed during redemption.

When users call `maxRedeem` to determine how many shares they can redeem, they will get an inflated number that includes both their actual shares and their rewards (which aren't shares). If they attempt to redeem this full amount, the transaction will revert because:

1. The `redeem` function validates that the user has sufficient shares.
2. The user doesn't actually own `balanceOf(owner)+ rewards[owner]` shares.
3. The rewards are already accounted for separately in the asset calculation during redemption.

This creates a confusing user experience where the contract returns a value from `maxRedeem` that cannot actually be used in a `redeem` call.

Recommendation:

Remove the override of the `maxRedeem` function to revert to the default ERC4626 implementation, which correctly returns only the shares balance. This ensures users receive an accurate value representing the maximum shares they can redeem without transaction failures.

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.4.6 Missing Slippage Control

Severity: *Low risk (Resolved)*

Context: ***AirDropStaking.sol***:*25*

Description:

The `AirDropStaking` contract implements ERC4626 vault functions (`deposit`, `withdraw`, `mint`, `redeem`) without explicit slippage protection parameters. In general DeFi applications, missing slippage protection can be a significant risk, but in this specific implementation, the risk profile is different due to the nature of the vault.

Unlike volatile AMM pools where front-running is common, this staking vault has several characteristics that naturally mitigate slippage concerns:

1. Predictable value accrual: The share price (exchange rate) in this vault changes primarily through reward distribution, which follows a predictable linear schedule and generally benefits existing shareholders.
2. Cooling period enforcement: The contract's cooling period (default 30 minutes) already creates a time barrier that reduces the impact of timing attacks.
3. Protected reserve accounting: The vault properly segregates staked assets from reward tokens in its accounting, ensuring reserve consistency.

With that being said, adding slippage parameters to all value-moving functions (`deposit`, `mint`, `withdraw`, `redeem`) would give users additional control over their acceptable exchange rates if desired.

Recommendation:

While not critical for this implementation, adding optional slippage protection would still follow best practices.

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.4.7 Incorrect Operation Order in mint Function

Severity: *Low risk (Resolved)*

Context: ***AirDropStaking.sol***:240-246

Description:

In the `mint` function of the `AirDropStaking` contract, the reward state update occurs after calculating the required assets, rather than before:

```
function mint(
    uint256 shares,
    address receiver
) public override(ERC4626, IERC4626) nonReentrant whenNotPaused returns (uint256
    assets) {
    assets = previewMint(shares);
    if (assets < minimumDepositAmount)
        IAirDropStaking.InvalidMinimumDeposit.selector.revertWith();
    _updateReward(receiver);  // Reward update happens after previewMint
    // ...
}
```

While `totalAssets()` may not be directly affected by reward updates (since rewards remain in the contract), the state inconsistency violates best practices for ERC4626 vaults where state should be

fully updated before performing asset/share calculations. In contrast, the `deposit` function correctly updates rewards before any asset/share calculations.

Recommendation:

Modify the `mint` function to update rewards before calculating assets. This ensures that all state is current before any asset calculations are performed, maintaining consistency with other functions and following best practices.

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.4.8 Circumvention of minimum requirement

Severity: *Low risk (Resolved)*

Context: ***AirDropStaking.sol***:*267-318*

Description:

The `withdraw` function in the `AirDropStaking` contract does not validate that users maintain the minimum threshold after a partial withdrawal.

While the `deposit` and `mint` functions enforce a minimum deposit amount:

```
if (assets < minimumDepositAmount)
    IAirDropStaking.InvalidMinimumDeposit.selector.revertWith();
```

This validation inconsistency creates a loophole where users can circumvent the minimum requirement by:

1. Initially depositing the required minimum amount (e.g., 1000 tokens)
2. Subsequently withdrawing most but not all of their funds (e.g., 999 tokens)
3. Leaving behind a dust amount (e.g., 1 token) that's below the minimum threshold

This undermines the contract's control mechanism and allows users to operate with unexpectedly small accounts, leading to rounding issues that can impact reward distributions.

Recommendation:

Consider adding validation to ensure that partial withdrawals don't leave behind balances below the minimum amount.

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.4.9 Unchecked Minimum Deposit Limit Update

Severity: *Low risk (Resolved)*

Context: ***AirDropStaking.sol***:*533-539*

Description:

The `setMinimumDepositAmount` function in the `AirDropStaking` contract fails to validate that the new minimum deposit amount does not exceed the current maximum deposit amount:

```
function setMinimumDepositAmount(
    uint256 newAmount
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (newAmount == 0) IAirDropStaking.InvalidAmount.selector.revertWith();
    emit MinimumDepositUpdated(minimumDepositAmount, newAmount);
    minimumDepositAmount = newAmount;
}
```

The function only validates that `newAmount` is not zero, but it doesn't check if `newAmount > maximumDepositAmount`. This creates a potential state inconsistency where the minimum deposit amount could be set higher than the maximum deposit amount.

In contrast, the `setMaximumDepositAmount` function correctly implements this validation with:

```
if (newAmount == 0 || newAmount < minimumDepositAmount)
    IAirDropStaking.InvalidAmount.selector.revertWith();
```

If an admin inadvertently sets the minimum deposit amount higher than the maximum, the contract would enter an unusable state where all deposits would revert. This is because the `deposit` and `mint` functions check that the asset amount is at least the minimum deposit amount, but the maximum deposit limit would make it impossible to satisfy this requirement.

Recommendation:

Add validation to ensure the new minimum deposit amount doesn't exceed the current maximum.

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.4.10 Unclear Reward Routing on ERC4626 Redemption

Severity: *Low risk (Acknowledged)*

Context: ***AirDropStaking.sol***:311

Description:

The `redeem` function in the `AirDropStaking` contract includes an implementation detail that may cause user confusion: when shares are redeemed, all accumulated rewards of the owner are sent to the receiver, regardless of who owns the shares.

```
function redeem(
    uint256 shares,
    address receiver,
    address owner
) public override(ERC4626, IERC4626) nonReentrant whenNotPaused returns (uint256
    assets) {
    // ...
    uint256 reward = rewards[owner];
    rewards[owner] = 0;
    assets = previewRedeem(shares);
    uint256 totalWithdrawal = assets + reward;
    // ...
    _withdraw(_msgSender(), receiver, owner, totalWithdrawal, shares);
}
```

All rewards are sent to the `receiver` address, which may be different from the `owner` who earned them, as the ERC4626 standard allows for flexible redemption destinations. Users may not anticipate this behavior and could inadvertently direct their rewards to unintended addresses.

Recommendation:

Consider adding clear documentation in code comments and user interface to explain how rewards are handled during redemption.

Resolution: Acknowledged by the team.

3.4.11 Admin Can End Sale Before Advertised End Time

Severity: *Low risk (Acknowledged)*

Context: ***SeedLaunchVesting.sol***:192

Description:

The `SeedLaunchVesting` contract allows the contract owner to prematurely end a token sale through the `endSale` function without any validation that the sale's advertised end time has passed:

```
function endSale(uint256 _scheduleId) external nonReentrant onlyOwner {
    if (_scheduleId >= allSchedules.length) {
        ISeedLaunchVesting.ErrorInvalidScheduleId.selector.revertWith();
    }
    Schedule storage s = allSchedules[_scheduleId];
    if (s.status != ScheduleStatus.ACTIVE) {
        ISeedLaunchVesting.ErrorScheduleNotActive.selector.revertWith();
    }
    s.status = ScheduleStatus.ENDED;
    s.statusChangeTime = block.timestamp;
    uint256 leftover = s.unsoldTokens;
    if (leftover > 0) {
        s.unsoldTokens = 0;
        vestedToken.safeTransfer(address(treasury), leftover);
    }
    emit SaleEnded(_scheduleId, leftover);
}
```

This design creates a discrepancy between user expectations and admin capabilities. Users participating in the sale may expect it to remain active until the advertised `saleEnd` timestamp, but the admin can terminate it at any point. Early termination could prevent users from participating in the sale during the timeframe they expected to have available.

Recommendation:

Add a time-based validation to the `endSale` function that ensures it can only be called after the advertised end time:

```
function endSale(uint256 _scheduleId) external nonReentrant onlyOwner {
    if (_scheduleId >= allSchedules.length) {
        ISeedLaunchVesting.ErrorInvalidScheduleId.selector.revertWith();
    }
    Schedule storage s = allSchedules[_scheduleId];
    if (s.status != ScheduleStatus.ACTIVE) {
```

```
        ISeedLaunchVesting.ErrorScheduleNotActive.selector.revertWith();
    }
    // Ensure sale has reached its advertised end time
    if (block.timestamp < s.saleEnd) {
        ISeedLaunchVesting.ErrorSaleNotEnded.selector.revertWith();
    }
    // Continue with ending the sale...
```

Alternatively, document this admin capability clearly in user-facing documentation to set proper expectations for sale participants.

Resolution: Acknowledged by the team.

3.4.12 Precision Loss in Vesting Calculations

Severity: *Low risk (Partially Resolved)*

Context: ***Vesting.sol***:*349-351*

Description:

In both the `Vesting` and `SeedLaunchVesting` contracts, there's an arithmetic issue where division operations are performed before multiplication, leading to minor precision loss due to integer division truncation:

```
uint256 perInstallment = leftover / s.installments;
uint256 unlockedInstallments = perInstallment * intervalsPassed;
```

Similarly, in `SeedLaunchVesting.sol`:

```
uint256 remainingTokens = p.amount - tgeTokens;
uint256 chunk = remainingTokens / s.installments;
return unlocked + (chunk * intervalsPassed);
```

This approach causes small rounding errors in each vesting period. The truncated tokens from earlier installments become available only in the final installment, slightly delaying the release of these tokens.

Recommendation:

Reorder the operations to perform multiplication before division to minimize precision loss. This approach ensures more accurate token distribution across vesting periods.

Resolution: Partially resolved by the team in **8abf9b1**

3.4.13 Inconsistent Rounding in Vesting Calculations

Severity: *Low risk (Resolved)*

Context: ***Vesting.sol***:*20*

Description:

The `Vesting` contract contains inconsistent rounding behavior across related functions that calculate vesting intervals. This discrepancy creates a mismatch between what users expect to receive and what they actually receive when claiming tokens.

In one function, intervals are rounded up (ceiling division):

```
uint256 intervalsPassed = (timePassed + s.installmentInterval - 1) / s.
    installmentInterval; // Ceiling division
```

While in the `getTotalUnlocked` function, intervals are rounded down (floor division):

```
uint256 intervalsPassed = timePassed / s.installmentInterval; // Floor division
```

This inconsistency leads to two specific issues:

1. When calling functions that estimate unlocked tokens, users may see higher amounts than they can actually claim due to the different rounding methods.
2. In the calculation for time until the next installment or remaining installments, the ceiling division causes incorrect behavior for the final installment.

This creates confusion for users who expect consistent behavior across the vesting contract's functions.

Recommendation:

Round down to ensure consistent behavior in the following functions: `getUnlockedInstallments`, `getNextInstallmentTimestamp`, `getRemainingInstallments`.

Resolution: Resolved. The recommended fix was implemented in **8abf9b1**.

3.4.14 Centralization Risk

Severity: *Low risk (Acknowledged)*

Context: ***Treasury.sol***:17

Description:

The protocol is subject to centralization risks stemming from powers granted to the admin. These powers include the ability to control the treasury which will initially hold all JACK tokens, control over pause functionality, and various other privileged function.

Recommendation:

Document these centralization risks thoroughly. This documentation should clearly outline the extent of admin powers and the potential impacts of their misuse.

Resolution: Acknowledged by the team.