

WalletConnect Contracts

Security Review

Cantina Managed review by:

Om Parikh, Security Researcher
Slowfi, Security Researcher

October 14, 2025

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 Medium Risk . . . 4
  - 3.1.1 Externally created locks can lead to DOS while claiming vesting allocation . . . 4
- 3.2 Low Risk . . . 4
  - 3.2.1 Incomplete Reward Distribution When More Than 52 Weeks Elapse Between Checkpoints 4
  - 3.2.2 Reward Injection Allowed After `kill()` and While Paused . . . 5
  - 3.2.3 Vesting revocation is not relayed to `StakeWeight` . . . 5
  - 3.2.4 Backdated Injections After Cursor Advance Cause Missed Rewards . . . 6
- 3.3 Gas Optimization . . . 7
  - 3.3.1 Immutable Variable Optimization . . . 7
  - 3.3.2 Redundant Check for Active Permanent Lock . . . 8
  - 3.3.3 Missing Reusable Pause Modifier Across Contracts . . . 8
- 3.4 Informational . . . 9
  - 3.4.1 Unreachable Branch in `_checkpointToken` Leads to Dead Code . . . 9
  - 3.4.2 Undistributed Injected Rewards Remain Stuck Until `kill()` . . . 9
  - 3.4.3 Consistent Casting With `SafeCast` . . . 10
  - 3.4.4 Missing `_disableInitializers` in constructor . . . 10
  - 3.4.5 Ambiguous offchain guarantees for `MerkleVester` . . . 10
  - 3.4.6 Missing beneficiary check in post claim handler . . . 11

4 Appendix **12**
- 4.1 Upgrade safety and storage integrity . . . 12

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

WalletConnect is the connectivity layer for the Financial Internet.

From Sep 28th to Oct 5th the Cantina team conducted a review of contracts on commit hash 299e7ba1. The team identified a total of **14** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	1	0
Low Risk	4	2	2
Gas Optimizations	3	2	1
Informational	6	4	2
Total	**14**	**9**	**5**

3 Findings

3.1 Medium Risk

3.1.1 Externally created locks can lead to DOS while claiming vesting allocation

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: If a lock is created on `StakeWeight` independently by transferring tokens and not via any linked vesters and same user also has vesting schedule but vested amount was not utilized for lock then, while withdrawing from vester it will revert in inequality comparison in `LockedTokenStaker.handlePostClaim` because it is not accounting for transferred tokens in the active lock.

Since the `handlePostClaim` always has to be triggered even if `LockedTokenStaker` is not used by beneficiary to create lock, there is no way to claim vesting given such scenario.

Recommendation: Locks which are created independently without using staker require funds to transferred, so that can be tallied in post claim handler. Also, relevant test for this scenario (before & after fix) should be added:

```
   uint256 lockedAmount = SafeCast.toUint256(lock.amount);
+ uint256 transferredAmount = lock.transferredAmount;

  // Check if there's enough unlocked tokens for the claim
- if (remainingAllocation < lockedAmount + claimAmount) {
+ if (remainingAllocation + transferredAmount < lockedAmount + claimAmount) {
      revert CannotClaimLockedTokens(remainingAllocation, lockedAmount, claimAmount);
  }
```

WalletConnect: Fixed in commit 1d15cdfb.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 Incomplete Reward Distribution When More Than 52 Weeks Elapse Between Checkpoints

Severity: Low Risk

Context: StakingRewardDistributor.sol#L225-L261

Description: The `_checkpointToken` function in `StakingRewardDistributor` distributes rewards proportionally across elapsed weeks since the last checkpoint. However, it limits iteration to `MAX_REWARD_ITERATIONS` (52). If more than 52 weeks pass between `lastTokenTimestamp` and the current `block.timestamp`, the loop exits early while `lastTokenTimestamp` and `lastTokenBalance` are updated to the current time and balance.

As a result, any unprocessed period beyond the 52-week limit is never recorded in `tokensPerWeek`, and the corresponding tokens become permanently undistributable. Future checkpoint calls will compute a zero delta because `lastTokenTimestamp` has already advanced to the current timestamp, effectively stranding the remaining rewards in the contract balance.

Proof of Concept: The following test demonstrates how `_checkpointToken` fails to distribute rewards if more than 52 weeks elapse between checkpoints. It advances the timestamp by 60 weeks, deposits tokens, and then calls `checkpointToken()`. The test verifies that only the first 52 weekly buckets receive tokens while the remaining weeks stay empty, resulting in stranded undistributed rewards.

```
function test_SRD_Over52Weeks_LosesRewards() external {
    uint256 startWeek = stakingRewardDistributor.lastTokenTimestamp();
    uint256 weeksElapsed = 60 weeks; // > MAX_REWARD_ITERATIONS (52)
    uint256 amount = 1_000_000e18;

    // Fund SRD directly (bypasses transfer restrictions and approvals)
    deal(address(l2wct), address(stakingRewardDistributor), amount);
    vm.warp(block.timestamp + weeksElapsed);
    stakingRewardDistributor.checkpointToken();

    // Assert: only first 52 buckets are filled; later ones remain zero
```

```
    uint256 sumAssigned = 0;
    uint256 filledWeeks = 0;
    for (uint256 i = 0; i <= weeksElapsed; i += 1 weeks) {
        uint256 weekTs = startWeek + i;
        uint256 w = stakingRewardDistributor.tokensPerWeek(weekTs);
        sumAssigned += w;
        if (w > 0) filledWeeks++;
    }
    // Expect only 52 filled weeks
    assertEq(filledWeeks, 52, "Only first 52 weeks should be filled");
    // Expect stranded remainder (sumAssigned < amount)
    assertLt(sumAssigned, amount, "Remainder beyond 52 weeks is stranded");

    // And internal cursors advanced as if fully processed
    assertEq(
        stakingRewardDistributor.lastTokenTimestamp(),
        block.timestamp,
        "lastTokenTimestamp incorrectly advanced to now"
    );
}
```

Recommendation: Consider to prevent advancing `lastTokenTimestamp` beyond the last fully processed week when the iteration limit is reached. Alternatively, allocate the undistributed remainder to the final processed week to ensure all tokens are accounted for.

If the 52-week iteration cap is kept for gas-safety reasons, consider to emit an event warning about truncated distributions and document the operational requirement to checkpoint at least once per year to prevent reward loss.

WalletConnect: Acknowledged. This scenario requires both (1) zero user claims and (2) zero reward injections for 52+ consecutive weeks. Given operational requirements include weekly reward injections, and any active user claiming triggers a checkpoint, this is a non-issue for a live protocol.

Cantina managed: Acknowledged by WalletConnect team.

3.2.2 Reward Injection Allowed After `kill()` and While Paused

Severity: Low Risk

Context: StakingRewardDistributor.sol#L579

Description: In `StakingRewardDistributor`, reward injection paths are not gated by lifecycle or pause state. As a result:

- Rewards can still be injected after the contract has been killed (i.e., when `isKilled` is set and `Killed()` has been emitted).
- Rewards can be injected while paused, since the injection flow does not check the module's paused state.

This creates operational inconsistencies: a killed distributor can continue accumulating undistributed balances, and a paused distributor can still receive new rewards even though distribution/claims are expected to be halted.

Recommendation: Consider to enforce lifecycle and pause guards on all reward-injection entry points so that injections revert when the distributor is killed and are disallowed while paused. Document the intended policy clearly (e.g., whether injections during pause should queue or be rejected).

WalletConnect: Fixed in commit 555566ee.

Cantina Managed: Fix verified.

3.2.3 Vesting revocation is not relayed to `StakeWeight`

Severity: Low Risk

Context: MerkleVester.sol#L3722

Description: In case of vesting being revoked or cancelled by benefactor, this information is not relayed / checkpointed to `StakeWeight` for positions created via `LockedTokenStaker`, so weight is not updated accordingly.

Given `forceWithdrawAll` can be done on `StakeWeight` on address-by-address basis, it possible to revoke manually. However, until this is performed it can allow voting or claiming rewards with incorrect weight during that window.

Recommendation:

- Consider adding more documentation for benefactor role and in what cases vesting might be revoked so it is transparent for users.
- If benefactor is controlled by same party, planned efforts should be made to make sure incorrect weight is not allowed to be misused.
- `isStakeWeightPaused()` check can be removed from `forceWithdrawAll` if deemed necessary to pause in case of vesting revocation so that admin can pause and have enough time to react for large set of addresses.

WalletConnect: We're not making changes to the MerkleVester contract, as it's deployed from Magna's UI. Agreed on the pause check to be removed so forceWithdrawAll can be executed during pause, see this being useful. Fixed in commit b591e4ff.

Cantina Managed: Fix verified.

3.2.4 Backdated Injections After Cursor Advance Cause Missed Rewards

Severity: Low Risk

Context: StakingRewardDistributor.sol#L462

Description: `StakingRewardDistributor` advances `weekCursorOf[user]` to the current (or next) week when a user claims. If rewards are later injected for a past week that the user's cursor has already passed, those rewards are not considered during subsequent claims for that user. Users who delayed claiming (so their cursor remained behind) can capture the backdated allocation, while users who claimed earlier cannot-creating uneven distribution and a potential incentive to avoid timely claiming.

Proof Of Concept: The following test demonstrates the behavior: it advances users' cursors via claims, then injects additional rewards for a previous week. Users who had already moved their cursors forward cannot claim the backdated rewards, while a user who delayed claiming captures the extra allocation.

```
function test_claim_patterns_backdated_week1_extra_favors_one_time() public {
    // Grant role
    vm.startPrank(users.admin);
    stakingRewardDistributor.grantRole(stakingRewardDistributor.REWARD_MANAGER_ROLE(), address(this));
    vm.stopPrank();

    // Prepare users and balances
    address u1 = makeAddr("u1b");
    address u2 = makeAddr("u2b");
    address u3 = makeAddr("u3b");
    deal(address(l2wct), u1, 10 ether);
    deal(address(l2wct), u2, 10 ether);
    deal(address(l2wct), u3, 10 ether);

    vm.prank(u1); l2wct.approve(address(stakeWeight), type(uint256).max);
    vm.prank(u2); l2wct.approve(address(stakeWeight), type(uint256).max);
    vm.prank(u3); l2wct.approve(address(stakeWeight), type(uint256).max);

    vm.prank(u1); stakeWeight.createPermanentLock(10 ether, 4 weeks);
    vm.prank(u2); stakeWeight.createPermanentLock(10 ether, 4 weeks);
    vm.prank(u3); stakeWeight.createPermanentLock(10 ether, 4 weeks);

    deal(address(l2wct), address(this), 100 ether);
    l2wct.approve(address(stakingRewardDistributor), type(uint256).max);

    uint256 t0 = (block.timestamp / 1 weeks) * 1 weeks;
    vm.warp(t0);

    // Base rewards: 1 ether for week1 and week2
    stakingRewardDistributor.injectReward(t0 + 1 weeks, 1 ether);
    stakingRewardDistributor.injectReward(t0 + 2 weeks, 1 ether);

    uint256 totalU1 = 0; uint256 totalU2 = 0; uint256 totalU3 = 0;

    // U2 mid-week1 claim (zero)
```

```
    vm.warp(t0 + (1 weeks)/2); vm.prank(u2); totalU2 += stakingRewardDistributor.claim(u2);

    // End of week1 effective claim point t0+2w: U1 and U2 claim week1
    vm.warp(t0 + 2 weeks); vm.prank(u1); totalU1 += stakingRewardDistributor.claim(u1);
    vm.prank(u2); totalU2 += stakingRewardDistributor.claim(u2);

    // Mid of week3: backdate extra to week1 after U1/U2 have moved past
    vm.warp(t0 + 2 weeks + (1 weeks)/2);
    stakingRewardDistributor.injectReward(t0 + 1 weeks, 1 ether); // extra late for week1
    vm.prank(u2); totalU2 += stakingRewardDistributor.claim(u2); // still zero

    // End of week2 effective claim point t0+3w: all claim
    vm.warp(t0 + 3 weeks);
    vm.prank(u1); totalU1 += stakingRewardDistributor.claim(u1);
    vm.prank(u2); totalU2 += stakingRewardDistributor.claim(u2);
    vm.prank(u3); totalU3 += stakingRewardDistributor.claim(u3);

    // u3 should be strictly greater; u1 and u2 should match each other
    assertEq(totalU1, totalU2, "u1 and u2 (who claimed week1 at end) should match");
    assertGt(totalU3, totalU1, "one-time claimer should capture backdated week1 extra");
}
```

Recommendation: Consider to disallow backdated reward injections (e.g., require `timestamp >= currentWeek`) or to handle backdated injections by re-opening past weeks for all users (e.g., allowing a bounded rescan window) so that users who claimed earlier do not miss subsequent allocations. If keeping the current behavior for gas/operational reasons, consider to document the behavior and enforce from the UI that claims are only enabled when they would collect rewards.

WalletConnect: Acknowledged, we will still keep the current functionality in case we missed injecting a week's rewards, as our front end is blocking to call the claim function if it will not return rewards. I see more harm on not being able to do this for our regular users in case we miss injecting rewards, than the potentially outlined case.

Cantina Managed: Acknowledged by WalletConnect team.

3.3 Gas Optimization

3.3.1 Immutable Variable Optimization

Severity: Gas Optimization

Context: LockedTokenStaker.sol#L61-L62, StakeWeight.sol#L242-L243, StakeWeight.sol#L248, StakingRewardDistributor.sol#L154, StakingRewardDistributor.sol#L157, StakingRewardDistributor.sol#L160

Description: Several global state variables across the contracts are assigned only once during `initialize()` and have no setters that modify them afterward. These variables behave as constants for the lifetime of the contract and can be declared as `immutable` on the implementation contract.

Using `immutable` variables stores their values directly in the contract bytecode, allowing access through a single `PUSH` operation instead of an `SLOAD`. This optimization reduces gas costs and can slightly decrease stack pressure in functions that frequently access these variables.

This change is only safe if the deployment model does not rely on proxy initialization (i.e., the values are passed in the constructor or fixed at implementation deploy time). In a proxy-based design where each proxy requires unique initialization parameters, these variables must remain in storage.

The following variables qualify for conversion to `immutable` under a non-proxy or single-instance deployment model:

- `StakingRewardDistributor.sol`: `config`, `emergencyReturn` and `startWeekCursor`.
- `StakeWeight.sol`: `config`.
- `LockedTokenStaker.sol`: `vesterContract` and `config`.

These variables are only set in the initializer and remain unchanged afterward. Variables related to roles or governance (e.g., `AccessControl` roles) are mutable and therefore excluded.

Recommendation: Consider to declare the following variables as `immutable` if the deployment model allows setting them via constructor arguments rather than an initializer:

- `config`, `emergencyReturn`, and `startWeekCursor` in `StakingRewardDistributor`.
- `config` in `StakeWeight`.
- `vesterContract` and `config` in `LockedTokenStaker`.

If upgradeable proxies will be used, keep them as storage variables but consider caching frequently accessed configuration fields into local variables in gas-critical functions to minimize repeated `SLOAD` operations.

WalletConnect: These contracts are deployed behind `TransparentUpgradeableProxy` and rely on `initialize()` to set per-instance state. Making `config`, `emergencyReturn`, or `vesterContract` immutable would:

- Break the proxy pattern: Immutables are set in the implementation constructor and baked into bytecode. Proxy storage is only written during `initialize()` via `delegatecall`, not during the constructor.
- Prevent multi-instance deployments: We deploy multiple `LockedTokenStaker` proxies (reown, walletconnect, backers) pointing to the same implementation but with different `vesterContract` addresses. If `vesterContract` were immutable, all proxies would share the same value.
- Complicate upgrades: Each upgrade would require a new implementation with different constructor arguments rather than simply passing different init parameters to the proxy.

Cantina managed: Acknowledged by WalletConnect team.

3.3.2 Redundant Check for Active Permanent Lock

Severity: Gas Optimization

Context: StakeWeight.sol#L1116

Description: In `StakeWeight::_withdrawExpiredLock`, the line

```
if (s.isPermanent[user]) revert LockStillActive(type(uint256).max);
```

is redundant because `_withdrawAll` which is called immediately afterwards, already performs the same validation. The repeated check does not affect correctness but slightly increases bytecode size and gas usage during execution.

Recommendation: Consider to remove the redundant `isPermanent[user]` check from `_withdrawExpiredLock` to simplify the logic and reduce gas cost, since the condition is already enforced by `_withdrawAll`.

WalletConnect: Fixed in commit 21e3ec63.

Cantina Managed: Fix verified.

3.3.3 Missing Reusable Pause Modifier Across Contracts

Severity: Gas Optimization

Context: StakeWeight.sol#L589, StakeWeight.sol#L603, StakeWeight.sol#L628, StakeWeight.sol#L793, StakeWeight.sol#L806, StakeWeight.sol#L823, StakeWeight.sol#L843, StakeWeight.sol#L866, StakeWeight.sol#L930, StakeWeight.sol#L972, StakeWeight.sol#L1126, StakeWeight.sol#L1177, StakeWeight.sol#L1328, StakeWeight.sol#L1423

Description: Contract `StakeWeight` perform the pause check inline using statements such as:

```
if (Pauser(config.getPauser()).isStakingRewardDistributorPaused()) revert Paused();
```

This logic independently, which increases code duplication, slightly raises stack usage, and makes the pausing mechanism harder to maintain if its interface evolves. As mentioned in "Immutable Variable Optimization", making `config` immutable allows implementing efficient reusable modifiers that perform these checks without redundant storage reads.

Recommendation: Consider to introduce shared pause-check modifiers (e.g., `whenNotPaused`) or internal helper functions that reference the immutable `config` variable. This would unify pause enforcement logic across all modules, improve maintainability, and slightly reduce gas and stack overhead.

WalletConnect: Fixed in commit 7516559b.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Unreachable Branch in `_checkpointToken` Leads to Dead Code

Severity: Informational

Context: StakingRewardDistributor.sol#L275-L276

Description: In `StakingRewardDistributor._checkpointToken`, the branch:

```
if (deltaSinceLastTimestamp == 0 && nextWeekCursor == timeCursor) {
    tokensPerWeek[thisWeekCursor] = tokensPerWeek[thisWeekCursor] + toDistribute;
}
```

is unreachable. By construction:

- `deltaSinceLastTimestamp == 0` implies `timeCursor == block.timestamp`.
- `thisWeekCursor = _timestampToFloorWeek(timeCursor)`.
- `nextWeekCursor = thisWeekCursor + 1 weeks`.

For any `timeCursor`, `nextWeekCursor` equals `floorWeek(timeCursor) + 1 weeks`, which is strictly greater than `timeCursor` (if `timeCursor` is on a week boundary, `floorWeek(timeCursor) == timeCursor` and `nextWeekCursor == timeCursor + 1 weeks`). Therefore, `nextWeekCursor == timeCursor` can never hold, making the branch dead code.

Impact is mainly maintainability and minor cognitive overhead (readers may assume a scenario that cannot occur). In some compilers/optimizations this also prevents the block from being optimized away at the source level.

Proof of Concept: Add the following property-style assertion around the computation to demonstrate impossibility (for review/testing only):

```
// Pseudo-instrumentation mirroring the function's variables
uint256 timeCursor = lastTokenTimestamp;
uint256 deltaSinceLastTimestamp = block.timestamp - timeCursor;
uint256 thisWeekCursor = _timestampToFloorWeek(timeCursor);
uint256 nextWeekCursor = thisWeekCursor + 1 weeks;

// When delta == 0 (e.g., call twice within same block), the equality cannot hold.
if (deltaSinceLastTimestamp == 0) {
    assert(nextWeekCursor != timeCursor); // always true
}
```

You can also reproduce `deltaSinceLastTimestamp == 0` by calling `checkpointToken()` twice in the same block; the assertion will never fail.

Recommendation: Consider to remove the unreachable branch or replace it with the intended edge-case handling (if any). If the aim was to "dump remainder into current week when no time elapsed," the correct condition would be solely `deltaSinceLastTimestamp == 0`, without comparing `nextWeekCursor` and `timeCursor`. Otherwise, delete the block and add a clarifying comment so future readers don't infer a non-existent path.

WalletConnect: Fixed in commit 1681da65.

Cantina Managed: Fix verified.

3.4.2 Undistributed Injected Rewards Remain Stuck Until `kill()`

Severity: Informational

Context: StakingRewardDistributor.sol#L581

Description: `StakingRewardDistributor` retains reward tokens internally and provides no mechanism to withdraw undistributed balances other than `kill()`, which forwards all remaining tokens to `emergencyReturn`. Tokens present in the contract but not mapped into `tokensPerWeek` are not claimable and remain

idle until the contract is killed. This does not result in a direct loss of funds but can cause temporary token stagnation.

Recommendation: Consider to document this behavior clearly so operators are aware that undistributed rewards remain locked within the contract until it is explicitly killed.

WalletConnect: Fixed in commit 86634970.

Cantina Managed: Fix verified.

3.4.3 Consistent Casting With `SafeCast`

Severity: Informational

Context: StakeWeight.sol#L1451

Description: The `StakeWeight.sol` contract casts a signed integer to an unsigned integer directly:

```
uint256 totalAmount = uint256(int256(lock.amount));
```

This direct cast bypasses the type-safety provided by `SafeCast` and can lead to unexpected behavior if the signed value were ever negative (even if such condition is currently prevented by logic elsewhere). Other sections of the codebase already use `SafeCast` for similar conversions, ensuring consistency and safety.

Recommendation: Consider to replace the manual cast with `SafeCast` (e.g., `SafeCast.toUint256(int256(lock.amount))`) for consistency and defensive safety. This maintains uniform coding standards across the codebase and provides a clearer guarantee against unintended sign conversion.

WalletConnect: Fixed in commit 497a1eae.

Cantina Managed: Fix verified.

3.4.4 Missing `_disableInitializers` in constructor

Severity: Informational

Context: StakeWeight.sol#L235

Description: Currently whenever an new implementation contract is deployed for `StakeWeight`, It can be initialized by anyone with unwanted values. This doesn't posses huge risk currently but should be prevented as a safety measure to avoid any issues if logic is upgraded in future which may increase risk.

Recommendation: `_disableInitializers()` should be added in `StakeWeight` to prevent unwanted initialization (or future reinitialization) on implementation contract.

WalletConnect: Fixed in commit 1575914a.

Cantina Managed: Fix verified.

3.4.5 Ambiguous offchain guarantees for `MerkleVester`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: There are certain things for `MerkleVester` which should hold true and guaranteed by offchain means as it is not checked on-chain:

- An instance of `MerkleVester` should only whitelist linked `LockedTokenStaker` as post claim handler at all times. If more addresses and`address(0)` or no address are whitelisted, `LockedTokenStaker` callback can be bypassed leading to missed accounting. This also implies a given `MerkleVester` should be exactly linked with one `LockedTokenStaker`.
- Intervals and calendar should have different allocation ids.
- A given beneficiary should have no more than one vesting schedule per `MerkleVester` . consider two allocations A1 (interval), A2 (calendar) on same vester where `A2.originalBenificary = A1.originalBenificary; A2.totalAllocation > A1.totalAllocation; A2.id != A1.id`, if lock is created from A1 and when benificiary tries to withdraw A2, it will revert.

Recommendation:

- Try implementing them on-chain wherever possible.
- Consider documenting them to relevant parties wherever needed.

WalletConnect: Acknowledged. We've reproduced and documented the reported behavior:

- `test/integration/generic/LockedTokenStaker.t.sol::test_MultiAllocation_WithActiveLock_OnDifferentAllocation`.
- `test/integration/concrete/locked-tokens-staker/handle-post-claim/handlePostClaimMultipleAllocations.t.sol`.

These tests confirm that if the same wallet receives two vesting allocations, a lock created from one can prevent withdrawals from the other until the lock expires. No funds are at risk—tokens remain in the vesting contract, but the user experiences a claim delay.

Mitigation:

The deployed contracts remain unchanged. We're addressing this through operational controls:

- WalletConnect will enforce ”one allocation per wallet” across all vesters that call `LockedTokenStaker`.
- We've requested Magna add a UI-level guardrail to flag/block duplicate wallets at allocation-creation time.

As long as this rule holds, the documented delay cannot occur. No production wallets will hit this path.

Tests commit: 3349955f.

Cantina Managed: Acknowledged.

3.4.6 Missing beneficiary check in post claim handler

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `createLockFor` and `increaseLockAmountFor` checks for `msg.sender` being beneficiary.

```
if (allocation.originalBeneficiary != msg.sender) {
    revert InvalidCaller();
}
```

However, it is not validated in `handlePostClaim` or in `MerkleVester`. Currently, the impact is trivial if someone else calls it passing leaf data as it will revert if lock is not expired or if expired, it will send funds to beneficiary's address. Worst case impact being if beneficiary wanted to change address to receive on new address but it can be forced to receive on same address.

Recommendation: Consider adding check in `MerkleVester` for `msg.sender` or in post claim handler for `tx.origin`.

WalletConnect: Acknowledged. We're not making changes to the MerkleVester contract, as it's deployed from Magna's UI.

Cantina Managed: Acknowledged.

4 Appendix

4.1 Upgrade safety and storage integrity

Currently, mainnet for tests ensures storage is preserved and works as expected after upgrade. However, the test only touches certain storage slots. Any collision or unwanted behaviour is not identified for those slots.

As, compiler parameters were changed from `evm:paris + runs:10_000` → `evm:cancun + runs:200` and storage layout is at slightly different location from ERC-7201, A sort of formal correctness can be obtained via dumping address account storage from op-reth / op-geth at current block and matching all slots after upgrade on foundry / tenderly fork. The runtime for comparison should be significantly less as it doesn't involve any network / rpc calls.

A sparse sample set for distinct user address and old blocks / epochs should be picked to check after upgrading, all historical API returns same result and new logic has not impacted any old behaviour.

While performinig upgrade on mainnet, relevant contracts should be paused or upgrade process should be atomic to avoid any synchronization issues.

WalletConnect:

- Restored `optimizer_runs = 10_000` to match deployed contracts (0xc746f9a45a06cbf2ad761442821c91a479151cc3).
- OpenZeppelin plugin validates storage compatibility (`StakeWeightPermanentUpgradeFork.t.sol:117-147`).
  - Fork tests verify state preservation with real mainnet positions.
  - All 5 core upgrade tests pass.

Will follow your recommendation with an atomic upgrade via timelock.

Note: `evm_version` upgrade (Paris Cancun) is safe - only affects opcodes, not storage. Required for deployed MerkleVester interactions fork tests.

Commit: 26c0bbfe

Added storage baseline regression tests at commit 6ac72b92.

Per your recommendation, implemented:

- 10 real mainnet users across multiple lock states.
- Historical API checks at 5 time offsets (0-52 weeks).
- Pre/post-upgrade baseline assertions.
- Fuzz testing for regression detection.

All historical queries return identical results post-upgrade

Cantina Managed: Fix verified. We agree EVM version update should be fine. `via-ir` also seems to be off in both cases (previous config and current config)