MyCut Protocol Audit Report

Version 1.0

Alchemist

January 28, 2026

MyCut Protocol Audit Report

Aditya Kumar Mishra

Jan 28, 2026

MyCut Protocol Audit Report

Prepared by: Team Alchemist

Lead Auditors: - Aditya Kumar Mishra

Assisting Auditors: - None

Table of contents

- High
- Medium
- Low
- Informational

About Alchemist

We are Team Alchemist, currently a one-member team founded by Aditya Kumar Mishra. Our focus is on learning, researching, and implementing smart contract security, with an emphasis on understanding vulnerabilities, best practices, and secure blockchain development.

Disclaimer

The Alchemist team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

The findings described in this document correspond to the following commit hash:

```
TBD
```

Scope

```
./src/
-- Pot.sol
-- ContestManager.sol
```

Protocol Summary

MyCut is a contest rewards distribution protocol which allows the setup and management of multiple reward distributions. The system allows authorized claimants 90 days to claim their allocated rewards before the contest manager takes a 10% cut of the remaining pool, with the remainder distributed equally among users who claimed in time. The protocol uses ERC20 tokens for rewards and implements a factory pattern through ContestManager to create individual Pot contracts for each contest.

Roles

- **Contest Manager Owner**: Deploys ContestManager contract, creates new contests, funds contests with ERC20 tokens, and can close contests after 90 days to claim manager cut.
- **Pot Owner**: The ContestManager contract that deploys each Pot, has exclusive rights to call `closePot()`.
- **Players/Claimants**: Users registered in a contest who can claim their allocated rewards during the 90-day claim window.
- **Manager**: Receives 10% of remaining unclaimed rewards after the 90-day period when `closePot()` is executed.

Executive Summary

This audit identified critical vulnerabilities in the MyCut protocol that enable denial of service attacks, permanent fund loss, and incorrect reward distributions. The most severe findings include unbounded loops causing gas exhaustion, division by zero errors, fund locking due to wrong array usage, manager cut being sent to contracts without withdrawal mechanisms, and duplicate player address handling that causes reward overwrites.

Issues found

Severity	Number of issues found
High	5
Medium	2
Low	1
Informational	0
Total	8

Findings

High

[H-1] Unbounded loop in `closePot()` causes permanent DoS and fund lock when many users claim

Description: The `closePot()` function iterates through the entire `claimants` array to distribute remaining rewards after the 90-day period. Since the `claimants` array grows unbounded with every successful `claimCut()` call, this loop can exceed the blockchain's block gas limit, making `closePot()` permanently uncallable and locking all remaining funds forever.

```
function closePot() external onlyOwner {
    if (block.timestamp - i_deployedAt < 90 days) {
        revert Pot__StillOpenForClaim();
    }
    if (remainingRewards > 0) {
        uint256 managerCut = remainingRewards / managerCutPercent;
        i_token.transfer(msg.sender, managerCut);

        uint256 claimantCut = (remainingRewards - managerCut) /
            i_players.length;
        // WARNING: UNBOUNDED LOOP - DoS when claimants.length is large
        for (uint256 i = 0; i < claimants.length; i++) {
            _transferReward(claimants[i], claimantCut);
        }
    }
}
```

Impact:

- CRITICAL severity - Protocol becomes permanently unusable
- Manager cannot collect their 10% cut

- Claimants cannot receive final distribution
- All `remainingRewards` permanently stuck in contract
- No emergency recovery mechanism exists

Proof of Concept:

Gas analysis from testing: - 2 claimants: 2,730 gas - 3 claimants: 20,862 gas - Rate: ~18,000 gas per claimant

Calculation:

```
Block gas limit = 30,000,000 gas
Max claimants before DoS ~= 1,666 claimants
With 2,000 claimants: ~36M gas required
Result: Transaction reverts with "out of gas"
```

Test results:

```
Claimants: 2    | Gas: 2,730
Claimants: 3    | Gas: 20,862
Claimants: 100  | Gas: ~1,800,000
Claimants: 500  | Gas: ~9,000,000
Claimants: 1000 | Gas: ~18,000,000
Claimants: 2000 | Gas: ~36,000,000 (EXCEEDS LIMIT)
```

Recommended Mitigation:

Implement pull-over-push pattern where users withdraw their distribution instead of contract pushing to all addresses in a single transaction:

```
+ mapping(address => uint256) public claimantDistribution;
+ mapping(address => bool) public hasClaimedDistribution;
+ bool public distributionReady;

function closePot() external onlyOwner {
    if (block.timestamp - i_deployedAt < 90 days) {
        revert Pot__StillOpenForClaim();
    }
    if (remainingRewards > 0) {
        uint256 managerCut = remainingRewards / managerCutPercent;
        i_token.transfer(msg.sender, managerCut);

        uint256 claimantCut = (remainingRewards - managerCut) /
            claimants.length;
-       for (uint256 i = 0; i < claimants.length; i++) {
-           _transferReward(claimants[i], claimantCut);
-       }
+
+       // Store distribution amount for each claimant
+       for (uint256 i = 0; i < claimants.length; i++) {
```

```
+            claimantDistribution[claimants[i]] = claimantCut;
+        }
+        distributionReady = true;
     }
}

+ // Let users pull their distribution
+ function withdrawDistribution() external {
+     require(distributionReady, "Distribution not ready");
+     require(!hasClaimedDistribution[msg.sender], "Already claimed
    distribution");
+     uint256 amount = claimantDistribution[msg.sender];
+     require(amount > 0, "No distribution available");
+
+     hasClaimedDistribution[msg.sender] = true;
+     _transferReward(msg.sender, amount);
+ }
```

[H-2] Manager cut sent to ContestManager contract with no withdrawal mechanism causes permanent fund lock

Description: When `closePot()` is called via `ContestManager.closeContest()`, the 10% manager cut is sent to `msg.sender`, which is the ContestManager contract itself. However, ContestManager has no function to withdraw ERC20 tokens, causing all manager cuts to be permanently locked in the ContestManager contract.

Root cause chain: 1. ContestManager creates Pot via `createContest()` -> Pot's owner = ContestManager address 2. ContestManager calls `Pot.closePot()` via `closeContest()` 3. Pot sends manager cut: `i_token.transfer(msg.sender, managerCut)` where `msg.sender = ContestManager` 4. ContestManager has no `withdraw()` or `rescue()` function 5. Funds permanently locked

```
// In Pot.sol
function closePot() external onlyOwner {
    // ...
    uint256 managerCut = remainingRewards / managerCutPercent;
    i_token.transfer(msg.sender, managerCut); // WARNING: Sends to
       ContestManager
    // ...
}

// In ContestManager.sol - NO WITHDRAWAL FUNCTION
contract ContestManager is Ownable {
    // ... no way to extract tokens sent to this contract
}
```

Impact:

- CRITICAL severity - Protocol economically broken
- 10% of remaining rewards permanently locked per contest
- No recovery mechanism exists
- Affects ALL contests created by the system
- Cumulative loss increases with each closed contest

Financial impact example:

```
Contest 1: 10,000 tokens  ->  1,000 locked (10%)
Contest 2: 50,000 tokens  ->  5,000 locked (10%)
Contest 3: 100,000 tokens  ->  10,000 locked (10%)
Total locked: 16,000 tokens permanently lost
```

Proof of Concept:

```
function testManagerCutLockedInContractManager() public {
    // Setup and fund contest
    vm.startPrank(owner);
    address pot = manager.createContest(players, rewards, token, 200e18
        );
    token.approve(address(manager), 200e18);
    manager.fundContest(0);

    // Warp 90 days and close
    vm.warp(block.timestamp + 91 days);

    uint256 managerBalanceBefore = token.balanceOf(address(manager));
    assertEq(managerBalanceBefore, 0);

    manager.closeContest(pot);

    // Manager cut (10% of 200 = 20) sent to ContestManager
    uint256 managerBalanceAfter = token.balanceOf(address(manager));
    assertEq(managerBalanceAfter, 20e18);

    // Owner did NOT receive the manager cut
    uint256 ownerBalance = token.balanceOf(owner);
    assertEq(ownerBalance, 800e18); // Still only has leftover from
        initial mint

    // NO FUNCTION TO WITHDRAW - funds stuck forever
    vm.stopPrank();
}
```

Recommended Mitigation:

Option 1: Send to actual owner (RECOMMENDED)

```
// In Pot.sol
function closePot() external onlyOwner {
    if (block.timestamp - i_deployedAt < 90 days) {
        revert Pot__StillOpenForClaim();
    }
    if (remainingRewards > 0) {
        uint256 managerCut = remainingRewards / managerCutPercent;
-       i_token.transfer(msg.sender, managerCut);
+       // Get the actual human owner (ContestManager's owner)
+       address actualOwner = Ownable(msg.sender).owner();
+       i_token.transfer(actualOwner, managerCut);

        // ... rest of code
    }
}
```

Option 2: Add withdrawal function to ContestManager

```
// In ContestManager.sol
+ function withdrawTokens(IERC20 token, address recipient, uint256 amount) external onlyOwner {
+     require(token.balanceOf(address(this)) >= amount, "Insufficient balance");
+     token.transfer(recipient, amount);
+ }
+
+ function rescueTokens(IERC20 token) external onlyOwner {
+     uint256 balance = token.balanceOf(address(this));
+     require(balance > 0, "No tokens to rescue");
+     token.transfer(msg.sender, balance);
+ }
```

[H-3] Division by zero and incorrect array usage causes DoS and fund loss

Description: The `closePot()` function has two critical flaws:

1. **Division by zero**: When no users claim rewards (`claimants.length = 0`), the function attempts to calculate `claimantCut = remaining / claimants.length`, causing a division by zero panic and making `closePot()` permanently uncallable.
2. **Wrong array length**: The function divides by `i_players.length` (all registered players) but distributes to `claimants.length` (only those who claimed), causing significant fund loss when fewer players claim than were registered.

```
function closePot() external onlyOwner {
    if (block.timestamp - i_deployedAt < 90 days) {
        revert Pot__StillOpenForClaim();
```

```
    }
    if (remainingRewards > 0) {
        uint256 managerCut = remainingRewards / managerCutPercent;
        i_token.transfer(msg.sender, managerCut);

        // WARNING: BUG 1: Uses i_players.length instead of claimants.length
        uint256 claimantCut = (remainingRewards - managerCut) / i_players.length;

        // WARNING: BUG 2: If claimants.length = 0, division by zero on line above
        for (uint256 i = 0; i < claimants.length; i++) {
            _transferReward(claimants[i], claimantCut);
        }
    }
}
```

Impact:

- HIGH severity for both issues
- Permanent DoS when no claimants (division by zero revert)
- Significant token loss when `claimants.length < i_players.length`
- Manager cannot collect their 10% cut
- All funds permanently locked in contract

Example scenarios:

Scenario 1: No claimants

```
10 players registered, 0 claimed
Try to close pot after 90 days
claimantCut = 1000 / 0  ->  PANIC (division by zero)
Result: All 1000 tokens stuck forever
```

Scenario 2: Wrong array length

```
10 players registered, 2 claimed
Remaining: 800 tokens
Manager cut: 80 tokens (10%)
Leftover: 720 tokens

BUG: claimantCut = 720 / 10 = 72 per person
Distributed: 72 * 2 = 144 tokens
Expected: 720 / 2 = 360 per person = 720 total
Lost: 720 - 144 = 576 tokens stuck (80% loss!)
```

Proof of Concept:

```
function test_Division_By_Zero_No_Claimants() public {
    // Create and fund contest
    vm.startPrank(owner);
    address pot = manager.createContest(players, rewards, token, 100);
    manager.fundContest(0);
    vm.stopPrank();

    // NO ONE CLAIMS (claimants.length = 0)

    vm.warp(block.timestamp + 91 days);

    // Try to close - reverts with division by zero
    vm.startPrank(owner);
    vm.expectRevert(); // Panic: division by zero
    manager.closeContest(pot);
    vm.stopPrank();

    // All 100 tokens stuck forever
}

function test_Wrong_Array_Fund_Loss() public {
    // 10 players registered, only 2 claim
    address[] memory manyPlayers = new address[](10);
    uint256[] memory manyRewards = new uint256[](10);

    for (uint256 i = 0; i < 10; i++) {
        manyPlayers[i] = makeAddr(string(abi.encodePacked("player", i))
            );
        manyRewards[i] = 10;
    }

    vm.startPrank(owner);
    address pot = manager.createContest(manyPlayers, manyRewards, token
        , 100);
    manager.fundContest(0);
    vm.stopPrank();

    // Only 2 players claim
    vm.prank(manyPlayers);
    Pot(pot).claimCut();

    vm.prank(manyPlayers); [ppl-ai-file-upload.s3.amazonaws](https://
        ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments
        /images/50523984/449e3f4d-9fad-462b-8c42-b08784ab76b6/image.jpg)
    Pot(pot).claimCut();

    vm.warp(block.timestamp + 91 days);

    vm.prank(owner);
    manager.closeContest(pot);

```

```
48      // Check how much is stuck
49      uint256 stuck = token.balanceOf(pot);
50      console.log("Tokens stuck:", stuck); // ~58 tokens stuck
51      assertGt(stuck, 50); // More than 50% of remaining funds lost
52  }
```

Recommended Mitigation:

```
1  function closePot() external onlyOwner {
2      if (block.timestamp - i_deployedAt < 90 days) {
3          revert Pot__StillOpenForClaim();
4      }
5      if (remainingRewards > 0) {
6          uint256 managerCut = remainingRewards / managerCutPercent;
7          i_token.transfer(msg.sender, managerCut);
8
9  +       if (claimants.length == 0) {
10 +           // No claimants, send all remaining to manager
11 +           i_token.transfer(msg.sender, remainingRewards - managerCut)
       ;
12 +       } else {
13 -           uint256 claimantCut = (remainingRewards - managerCut) /
       i_players.length;
14 +           uint256 claimantCut = (remainingRewards - managerCut) /
       claimants.length;
15             for (uint256 i = 0; i < claimants.length; i++) {
16                 _transferReward(claimants[i], claimantCut);
17             }
18 +       }
19     }
20 }
```

[H-4] Duplicate player addresses in constructor overwrite rewards causing fund loss

Description: The Pot constructor iterates through the `i_players` array and maps each address to their reward amount in `playersToRewards`. However, the constructor does not check for duplicate addresses. When a player address appears multiple times in the array, the mapping overwrites previous values, causing players to lose their rightful rewards and leaving funds permanently stuck in the contract.

```
1  constructor(address[] memory players, uint256[] memory rewards, IERC20
       token, uint256 totalRewards) {
2      i_players = players;
3      i_rewards = rewards;
4      i_token = token;
5      i_totalRewards = totalRewards;
6      remainingRewards = totalRewards;
7      i_deployedAt = block.timestamp;
```

```
8
9       // WARNING: NO CHECK FOR DUPLICATES
10      for (uint256 i = 0; i < i_players.length; i++) {
11          playersToRewards[i_players[i]] = i_rewards[i]; // Overwrites if
                duplicate
12      }
13  }
```

Impact:

- HIGH severity - Direct financial loss for users
- Players lose rightful rewards (only receive last occurrence)
- Funds permanently stuck in contract
- `totalRewards` accounting breaks
- `remainingRewards` never reaches zero even if all claim
- Manager receives less than expected cut

Example scenario:

```
1  players = [Alice, Bob, Alice]
2  rewards =
3
4  Loop iterations:
5  i=0: playersToRewards[Alice] = 100 [OK]
6  i=1: playersToRewards[Bob] = 200 [OK]
7  i=2: playersToRewards[Alice] = 50  WARNING: OVERWRITES 100!
8
9  Result:
10 - Alice can only claim 50 (lost 100!)
11 - Bob can claim 200
12 - Total claimable: 250
13 - Contract expects: 350
14 - 100 tokens stuck forever
```

Proof of Concept:

```
1  function test_Duplicate_Address_Overwrites_Reward() public {
2      address[] memory duplicatePlayers = new address[](3);
3      duplicatePlayers = player1;
4      duplicatePlayers = player2; [ppl-ai-file-upload.s3.amazonaws](https
           ://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/
           attachments/images/50523984/449e3f4d-9fad-462b-8c42-b08784ab76b6
           /image.jpg)
5      duplicatePlayers = player1; // Duplicate!
6
7      uint256[] memory duplicateRewards = new uint256[](3);
8      duplicateRewards = 100; // player1 first reward
9      duplicateRewards = 200; // player2 reward [ppl-ai-file-upload.s3.
           amazonaws](https://ppl-ai-file-upload.s3.amazonaws.com/web/
```

```
        direct-files/attachments/images/50523984/449e3f4d-9fad-462b-8c42
        -b08784ab76b6/image.jpg)
    duplicateRewards = 50;  // player1 second reward (overwrites first)

    vm.startPrank(owner);
    address pot = manager.createContest(duplicatePlayers,
        duplicateRewards, token, 350);
    manager.fundContest(0);
    vm.stopPrank();

    // Check what player1 can claim
    uint256 player1Reward = Pot(pot).checkCut(player1);
    assertEq(player1Reward, 50); // Only last entry, not 100!

    uint256 player2Reward = Pot(pot).checkCut(player2);
    assertEq(player2Reward, 200);

    // Player1 should get 150 (100+50) but only gets 50
    // 100 tokens permanently stuck in contract

    vm.prank(player1);
    Pot(pot).claimCut();

    vm.prank(player2);
    Pot(pot).claimCut();

    // Check remaining - should be 0 but is 100
    uint256 remaining = Pot(pot).getRemainingRewards();
    assertEq(remaining, 100); // 100 tokens stuck forever
}
```

Recommended Mitigation:

```
+ error Pot__DuplicatePlayer();
+ error Pot__ArrayLengthMismatch();

constructor(address[] memory players, uint256[] memory rewards, IERC20
    token, uint256 totalRewards) {
+   require(players.length == rewards.length, "Array length mismatch");
+
    i_players = players;
    i_rewards = rewards;
    i_token = token;
    i_totalRewards = totalRewards;
    remainingRewards = totalRewards;
    i_deployedAt = block.timestamp;

    for (uint256 i = 0; i < i_players.length; i++) {
+       // Check for duplicate addresses
+       if (playersToRewards[i_players[i]] != 0) {
+           revert Pot__DuplicatePlayer();
```

```
18 +        }
19 +        // Also check for zero address
20 +        require(i_players[i] != address(0), "Invalid player address");
21 +        require(i_rewards[i] > 0, "Reward must be greater than zero");
22          playersToRewards[i_players[i]] = i_rewards[i];
23      }
24  }
```

Alternative: Use a mapping to track seen addresses:

```
1  constructor(address[] memory players, uint256[] memory rewards, IERC20
       token, uint256 totalRewards) {
2      require(players.length == rewards.length, "Array length mismatch");
3
4  +   mapping(address => bool) memory seenPlayers;
5
6      i_players = players;
7      i_rewards = rewards;
8      i_token = token;
9      i_totalRewards = totalRewards;
10     remainingRewards = totalRewards;
11     i_deployedAt = block.timestamp;
12
13     for (uint256 i = 0; i < i_players.length; i++) {
14 +       require(!seenPlayers[i_players[i]], "Duplicate player address")
       ;
15 +       require(i_players[i] != address(0), "Zero address not allowed")
       ;
16 +       require(i_rewards[i] > 0, "Zero reward not allowed");
17 +
18 +       seenPlayers[i_players[i]] = true;
19         playersToRewards[i_players[i]] = i_rewards[i];
20     }
21 }
```

[H-5] Missing return value check on ERC20 transfer allows silent failures and fund loss

Description: The `_transferReward()` function uses `i_token.transfer()` without checking the return value. Some ERC20 tokens (like USDT) do not revert on failure but return **`false`** instead. This allows transfers to silently fail while the contract updates state as if they succeeded, leading to accounting mismatches and potential fund loss.

```
1  function _transferReward(address player, uint256 reward) internal {
2      i_token.transfer(player, reward); // WARNING: No return value check
3  }
4
5  function claimCut() public {
6      address player = msg.sender;
```

```
7       uint256 reward = playersToRewards[player];
8       if (reward <= 0) {
9           revert Pot__RewardNotFound();
10      }
11      playersToRewards[player] = 0;       // State updated
12      remainingRewards -= reward;          // State updated
13      claimants.push(player);              // State updated
14      _transferReward(player, reward);     // Transfer might fail silently
            !
15  }
```

Impact:

- HIGH severity - Silent fund loss
- State updated but tokens not transferred
- `remainingRewards` becomes inaccurate
- Users lose ability to retry failed claims
- Manager receives incorrect cut calculation
- No way to recover from silent failures

Proof of Concept:

```
1   // Mock token that returns false on transfer
2   contract BadERC20 is ERC20 {
3       constructor() ERC20("Bad", "BAD") {}
4
5       function transfer(address, uint256) public override returns (bool)
            {
6           return false; // Silent failure
7       }
8   }
9
10  function test_Silent_Transfer_Failure() public {
11      BadERC20 badToken = new BadERC20();
12
13      vm.startPrank(owner);
14      address pot = manager.createContest(players, rewards, IERC20(
            badToken), 100);
15      badToken.mint(pot, 100);
16      vm.stopPrank();
17
18      uint256 balanceBefore = badToken.balanceOf(player1);
19
20      vm.prank(player1);
21      Pot(pot).claimCut(); // Doesn't revert!
22
23      uint256 balanceAfter = badToken.balanceOf(player1);
24
25      // State updated but no tokens transferred
```

```
26      assertEq(balanceAfter, balanceBefore); // Still 0
27      assertEq(Pot(pot).checkCut(player1), 0); // Marked as claimed
28
29      // Player1 lost their reward forever!
30  }
```

Recommended Mitigation:

Use OpenZeppelin's `SafeERC20` library:

```
1  + import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/
       SafeERC20.sol";
2
3  contract Pot is Ownable {
4  +   using SafeERC20 for IERC20;
5
6       // ... rest of code
7
8       function _transferReward(address player, uint256 reward) internal {
9  -        i_token.transfer(player, reward);
10 +        i_token.safeTransfer(player, reward);
11      }
12
13      function closePot() external onlyOwner {
14          // ...
15          uint256 managerCut = remainingRewards / managerCutPercent;
16 -        i_token.transfer(msg.sender, managerCut);
17 +        i_token.safeTransfer(msg.sender, managerCut);
18          // ...
19      }
20 }
```

Or manually check return values:

```
1  function _transferReward(address player, uint256 reward) internal {
2  -   i_token.transfer(player, reward);
3  +   bool success = i_token.transfer(player, reward);
4  +   require(success, "Transfer failed");
5  }
```

Medium

[M-1] Lack of input validation in constructor allows invalid contract deployment

Description: The Pot constructor accepts `players` and `rewards` arrays without validating their lengths match or checking for empty arrays. This allows deployment of invalid Pot contracts that will fail when users try to interact with them, wasting gas and potentially locking funds if the contract is funded before the issue is discovered.

```
constructor(address[] memory players, uint256[] memory rewards, IERC20 token, uint256 totalRewards) {
    // WARNING: No validation
    i_players = players;
    i_rewards = rewards;
    i_token = token;
    i_totalRewards = totalRewards;
    remainingRewards = totalRewards;
    i_deployedAt = block.timestamp;

    for (uint256 i = 0; i < i_players.length; i++) {
        playersToRewards[i_players[i]] = i_rewards[i]; // Will revert if arrays mismatched
    }
}
```

Impact:

- MEDIUM severity - Wasted gas and poor UX
- Contract deployed with mismatched arrays causes index out of bounds
- Empty arrays create useless contests
- Zero totalRewards creates unintended behavior
- Zero address token causes all transfers to fail
- Funds sent to invalid contract are unrecoverable

Proof of Concept:

```
function test_Mismatched_Arrays() public {
    address[] memory players = new address[](2);
    players = player1;
    players = player2; [ppl-ai-file-upload.s3.amazonaws](https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/50523984/449e3f4d-9fad-462b-8c42-b08784ab76b6/image.jpg)

    uint256[] memory rewards = new uint256[](1); // Mismatch!
    rewards = 100;

    vm.startPrank(owner);
    // Deploys successfully but unusable
    address pot = manager.createContest(players, rewards, token, 100);

    // Funding works
    manager.fundContest(0);

    // But claims fail due to array mismatch
    vm.prank(player2);
    vm.expectRevert(); // Array index out of bounds
    Pot(pot).claimCut();

```

```
21      // Funds stuck in unusable contract
22      vm.stopPrank();
23  }
```

Recommended Mitigation:

```
1  + error Pot__ArrayLengthMismatch();
2  + error Pot__EmptyArrays();
3  + error Pot__InvalidToken();
4  + error Pot__InvalidTotalRewards();
5
6  constructor(address[] memory players, uint256[] memory rewards, IERC20
       token, uint256 totalRewards) {
7  +   // Validate array lengths
8  +   require(players.length == rewards.length, "Array length mismatch");
9  +   require(players.length > 0, "Empty players array");
10 +
11 +   // Validate token
12 +   require(address(token) != address(0), "Invalid token address");
13 +
14 +   // Validate totalRewards
15 +   require(totalRewards > 0, "Total rewards must be greater than zero"
       );
16 +
17 +   // Calculate expected total from rewards array
18 +   uint256 calculatedTotal = 0;
19 +   for (uint256 i = 0; i < rewards.length; i++) {
20 +       calculatedTotal += rewards[i];
21 +   }
22 +   require(calculatedTotal == totalRewards, "Total rewards mismatch");
23
24      i_players = players;
25      i_rewards = rewards;
26      i_token = token;
27      i_totalRewards = totalRewards;
28      remainingRewards = totalRewards;
29      i_deployedAt = block.timestamp;
30
31      for (uint256 i = 0; i < i_players.length; i++) {
32 +        require(i_players[i] != address(0), "Invalid player address");
33 +        require(i_rewards[i] > 0, "Reward must be greater than zero");
34          playersToRewards[i_players[i]] = i_rewards[i];
35      }
36  }
```

[M-2] Centralisation risk with single owner control and no timelock

Description: Both Pot and ContestManager contracts use OpenZeppelin's `Ownable` with a single owner having unrestricted control over critical functions. There is no multi-sig requirement, timelock,

or governance mechanism. A compromised or malicious owner can immediately close contests, withdraw funds, or create malicious contests without any delay or oversight.

Impact:

- MEDIUM severity - Trust assumption required
- Single point of failure
- No protection against compromised owner key
- Immediate fund extraction possible
- Users have no warning before critical changes
- No community governance or oversight

Proof of Concept:

```
function test_Owner_Can_Close_Early_And_Steal() public {
    // Compromised owner scenario
    vm.startPrank(owner);

    address pot = manager.createContest(players, rewards, token, 100);
    manager.fundContest(0);

    // Malicious owner waits 91 days and immediately closes
    // Users have no warning or time to claim
    vm.warp(block.timestamp + 91 days);

    // Owner closes instantly and takes 10% cut
    manager.closeContest(pot);

    // No timelock, no multi-sig, no governance
    vm.stopPrank();
}
```

Recommended Mitigation:

Implement multi-sig and timelock:

```
+ import "@openzeppelin/contracts/governance/TimelockController.sol";

- contract ContestManager is Ownable {
+ contract ContestManager {
+   TimelockController public timelock;
+   uint256 public constant MIN_DELAY = 2 days;

+   constructor(address[] memory proposers, address[] memory executors)
    {
+       timelock = new TimelockController(
+           MIN_DELAY,
+           proposers,
+           executors,
```

```
+            address(0)
+        );
+    }

+    modifier onlyTimelock() {
+        require(msg.sender == address(timelock), "Not timelock");
+        _;
+    }

-    function closeContest(address contest) public onlyOwner {
+    function closeContest(address contest) public onlyTimelock {
         _closeContest(contest);
     }
}
```

Or implement role-based access control:

```
+ import "@openzeppelin/contracts/access/AccessControl.sol";

- contract ContestManager is Ownable {
+ contract ContestManager is AccessControl {
+   bytes32 public constant MANAGER_ROLE = keccak256("MANAGER_ROLE");
+   bytes32 public constant CLOSER_ROLE = keccak256("CLOSER_ROLE");

+   constructor() {
+       _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
+       _grantRole(MANAGER_ROLE, msg.sender);
+   }

-   function createContest(...) public onlyOwner returns (address) {
+   function createContest(...) public onlyRole(MANAGER_ROLE) returns (address) {
        // ...
    }

-   function closeContest(address contest) public onlyOwner {
+   function closeContest(address contest) public onlyRole(CLOSER_ROLE) {
        _closeContest(contest);
    }
}
```

Low

[L-1] Missing events for critical state changes reduces transparency

Description: The protocol lacks events for important state changes such as contest creation, funding, and closure. Events are crucial for off-chain monitoring, indexing, and providing transparency to users.

Without events, it's difficult to track contest lifecycle and user actions.

Impact:

- LOW severity - Poor observability
- Difficult to track contest status off-chain
- No notification mechanism for users
- Hard to build frontend interfaces
- Reduced transparency for auditing

Recommended Mitigation:

```
contract ContestManager is Ownable {
+    event ContestCreated(address indexed contest, address indexed creator, uint256 totalRewards);
+    event ContestFunded(address indexed contest, uint256 amount);
+    event ContestClosed(address indexed contest, uint256 timestamp);

     function createContest(...) public onlyOwner returns (address) {
         Pot pot = new Pot(players, rewards, token, totalRewards);
         contests.push(address(pot));
         contestToTotalRewards[address(pot)] = totalRewards;
+        emit ContestCreated(address(pot), msg.sender, totalRewards);
         return address(pot);
     }

     function fundContest(uint256 index) public onlyOwner {
         Pot pot = Pot(contests[index]);
         IERC20 token = pot.getToken();
         uint256 totalRewards = contestToTotalRewards[address(pot)];

         if (token.balanceOf(msg.sender) < totalRewards) {
             revert ContestManager__InsufficientFunds();
         }

         token.transferFrom(msg.sender, address(pot), totalRewards);
+        emit ContestFunded(address(pot), totalRewards);
     }

     function _closeContest(address contest) internal {
         Pot pot = Pot(contest);
         pot.closePot();
+        emit ContestClosed(contest, block.timestamp);
     }
}

contract Pot is Ownable {
+    event RewardClaimed(address indexed player, uint256 amount);
+    event PotClosed(uint256 managerCut, uint256 claimantDistribution);

```

```
38      function claimCut() public {
39          // ... existing code ...
40          _transferReward(player, reward);
41  +       emit RewardClaimed(player, reward);
42      }
43
44      function closePot() external onlyOwner {
45          // ... existing code ...
46  +       emit PotClosed(managerCut, claimantCut);
47      }
48  }
```

Informational

No informational findings identified.

‘‘‘
