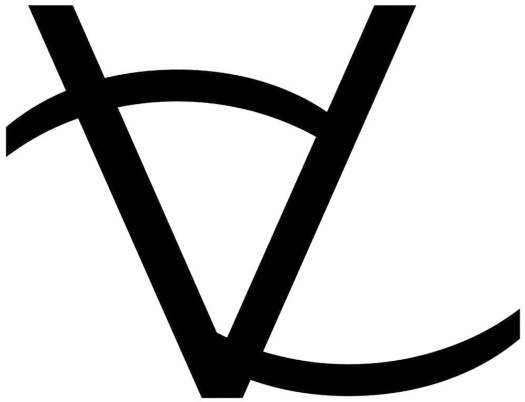

MyCut Security Review

Initial Version

Valya Zaitseva

January 18, 2026

MyCut Security Review

Valya Zaitseva

January 16, 2026

Prepared by: Valya Zaitseva

Table of Contents

* [H-5] Tokens are trapped in `Pot.sol` forever because no function allow to withdraw them
* [H-6] Possible discrepancy between sum of all rewards and `totalRewards` causes incorrect accounting and may lead to underflow error
* [H-7] No protocol profit and it may be economically impractical to close a contest
* [M-1] Unbound array of players may cause `OutOfGas` error and `DoS` attack
* [M-2] Possible inconsistency in `players` and `rewards` arrays length prevents contest creation
* [L-1] The return value of `ERC20:transfer()` and `ERC20:transferFrom()` functions in never evaluated which may lead to silent fail of token transfer.

Protocol Summary

MyCut is a contest rewards distribution protocol which allows the set up and management of multiple rewards distributions, allowing authorized claimants 90 days to claim before the manager takes a cut of the remaining pool and the remainder is distributed equally to those who claimed in time!

Disclaimer

As a solo reviewer, I make every effort to identify as many vulnerabilities in the code within the given time period, but hold no responsibility for the findings provided in this document. A security audit 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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

- src/
- *ContestManager.sol*
- *Pot.sol*

Roles

Owner/Admin (Trusted) - Is able to create new Pots, close old Pots when the claim period has elapsed and fund Pots User/Player - Can claim their cut of a Pot

Executive Summary

Issues found

Severity	Number of issues found
High	7
Medium	2
Low	1

Findings

High

[H-1] `Pot::remainingRewards` should be equal to a balance of `Pot.sol` as it reflects the internal accounting of a protocol

- *Impact: High*

- *Likelihood: High*

Root + Impact

Description

- The protocol's invariant `remainingRewards should be equal to pot balance` proves a protocol solvency.
- When a Pot is created, the value of `remainingRewards` is assigned equal to `totalRewards`. When a Pot is funded, exactly `totalRewards` amount is transferred to `Pot.sol`. This means that `remainingRewards` reflects `balanceOf(pot)` and these two values should be synchronized.
- A pot may be funded several times, but `remainingRewards` is not synchronized with a new balance.
- `remainingRewards` value is not assigned to zero or synchronized with a pot balance when `Pot::closePot()` function is called.

`Pot.sol` constructo assignes `remainingRewards` equal to `totalRewards` in constructor:

```
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          // i_token.transfer(address(this), i_totalRewards);
10         for (uint256 i = 0; i < i_players.length; i++) {
11             playersToRewards[i_players[i]] = i_rewards[i];
12         }
13     }
```

`ContestManager::fundContest` transfers `totalRewards` and sets a `Pot.sol` balance:

```
1  function fundContest(uint256 index) public onlyOwner {
2          Pot pot = Pot(contests[index]);
3          IERC20 token = pot.getToken();
4          uint256 totalRewards = contestToTotalRewards[address(pot)];
5
6          if (token.balanceOf(msg.sender) < totalRewards) {
7              revert ContestManager__InsufficientFunds();
8          }
9  @>        token.transferFrom(msg.sender, address(pot), totalRewards);
10     }
```

`remainingRewards` is not zeroed or synchronized with a pot balance, allowing it to be more than balance and causing a protocol insolvency:

```
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;
            for (uint256 i = 0; i < claimants.length; i++) {
                _transferReward(claimants[i], claimantCut);
            }
        }
    }
```

Risk

Likelihood:

- The issue occurs at the very beginning of the contest, when not-funded pot already has `remainingRewards` which means that a contest is ready to be started, but it was not funded yet and it is insolvent.
- Once a pot is funded, the invariant `remainingRewards should be equal to pot balance` holds till a `Pot::closePot()` function is called - `remainingRewards` is never zeroed or synchronized with a pot balance.

Impact:

- The issue breaks invariant `remainingRewards should be equal to pot balance` and reveals a protocol insolvency when remainingRewards exceed balance, which mean that it will not be possible to pay claimants cuts.

Proof of Concept: Values of `remainingRewards` and `Pot.sol` balance are not equal during a protocol lifecycle. Please, add the following test to `TestMyCut.t.sol`:

```
function test_remainingRewardsIsNotEqualToPotBalance() public
    mintAndApproveTokens {
        vm.startPrank(user);
        rewards = [500, 500];
        totalRewards = 1000;
        contest = ContestManager(conMan).createContest(players, rewards
            , IERC20(ERC20Mock(weth)), totalRewards);

        uint256 potBalInitial = ERC20Mock(weth).balanceOf(contest);
```

```
8           uint256 remainingRewardsInitial = Pot(contest).
                getRemainingRewards();
9           // remainingRewards != pot balance
10          assertGt(remainingRewardsInitial, potBalInitial);
11
12          ContestManager(conMan).fundContest(0);
13          vm.stopPrank();
14
15          uint256 potBalFunded = ERC20Mock(weth).balanceOf(contest);
16          uint256 remainingRewardsFunded = Pot(contest).
                getRemainingRewards();
17          // remainingRewards == pot balance
18          assertEq(potBalFunded, remainingRewardsFunded);
19
20          vm.startPrank(player1);
21          Pot(contest).claimCut();
22          vm.stopPrank();
23
24          uint256 potBalClaimed = ERC20Mock(weth).balanceOf(contest);
25          uint256 remainingRewardsClaimed = Pot(contest).
                getRemainingRewards();
26          // remainingRewards == pot balance
27          assertEq(potBalClaimed, remainingRewardsClaimed);
28
29          vm.warp(91 days);
30
31          vm.startPrank(user);
32          ContestManager(conMan).closeContest(contest);
33          vm.stopPrank();
34
35          uint256 potBalFinal = ERC20Mock(weth).balanceOf(contest);
36          uint256 remainingRewardsFinal = Pot(contest).
                getRemainingRewards();
37          // remainingRewards != pot balance
38          assertGt(remainingRewardsFinal, potBalFinal);
39      }
```

Recommended Mitigation: It is recommended to synchronize `remainingRewards` with a pot balance and account additionally transferred funds in `Pot::remainingRewards`:

```
-uint256 private remainingRewards;

constructor(address[] memory players, uint256[] memory rewards, IERC20
    token, uint256 totalRewards) {
     i_players = players;
     i_rewards = rewards;
     i_token = token;
     i_totalRewards = totalRewards;
-     remainingRewards = totalRewards;
     i_deployedAt = block.timestamp;

```

```
11      // i_token.transfer(address(this), i_totalRewards);
12      for (uint256 i = 0; i < i_players.length; i++) {
13          playersToRewards[i_players[i]] = i_rewards[i];
14      }
15  }
16
17  function closePot() external onlyOwner {
18      if (block.timestamp - i_deployedAt < 90 days) {
19          revert Pot__StillOpenForClaim();
20      }
21  +   uint256 remainingRewards = i_token.balanceOf(address(this));
22      if (remainingRewards > 0) {
23          uint256 managerCut = remainingRewards / managerCutPercent;
24          i_token.transfer(msg.sender, managerCut);
25          uint256 claimantCut = (remainingRewards - managerCut) /
                i_players.length;
26          for (uint256 i = 0; i < claimants.length; i++) {
27              _transferReward(claimants[i], claimantCut);
28          }
29      }
30  }
```

[H-2] `Pot::closePot()` may be called several times causing inflation of pot balance and players balance

- *Impact: High*
- *Likelihood: High*

Root + Impact

Description:

- When the contest is considered as done, the owner calls `Pot::closePot()` and the rewards should be distributed among claimants and a contest manager, and it is logically correct that the `remainingRewards` should be assigned to zero, but this never happens.

```
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   @>          uint256 claimantCut = (remainingRewards - managerCut) /
        i_players.length;
10              for (uint256 i = 0; i < claimants.length; i++) {
11                  _transferReward(claimants[i], claimantCut);
```

```
12                 }
13             }
14         }
```

Risk

Likelihood:

- The issue occurs because `remainingRewards` value is not zeroed when `Pot::closePot()` is called.

Impact:

- It it possible to call `Pot::closePot()` several times and the same amount of `remainingRewards` is used to calculate `managerCut` and `claimantCut`, so a manager and claimants receive extra funds and this may be done till there is enough tokens in the Pot.*
- Also, it is possible to call `Pot::claimCut()` after `Pot::closePot()` and receive a reward back because `remainingRewards` is not zeroed.

Proof of Concept: Please, add the following test `test_closeContestSeveralTimes` to `TestMyCut.t.sol`:

```
address player3 = makeAddr("player3");
address player4 = makeAddr("player4");
address[] playersLocal = [player1, player2, player3, player4];

    function test_closeContestSeveralTimes_callClaimAfterCloseContest()
        public mintAndApproveTokens {
        vm.startPrank(user);
        rewards = [500, 500, 500, 500];
        totalRewards = 2000;

        contest = ContestManager(conMan).createContest(playersLocal,
            rewards, IERC20(ERC20Mock(weth)), totalRewards);
        ContestManager(conMan).fundContest(0);
        vm.stopPrank();

        // one player claimed
        vm.startPrank(player1);
        Pot(contest).claimCut(); // 3 rewards are left
        vm.stopPrank();

        vm.warp(91 days);

        // close contest first time
        uint256 conManBalanceBeforeCloseContest1 = ERC20Mock(weth).
            balanceOf(conMan);
        uint256 playerBalanceBeforeCloseContest1 = ERC20Mock(weth).
            balanceOf(player1);
```

```
24          vm.prank(user);
25          ContestManager(conMan).closeContest(contest);
26          uint256 conManBalanceAfterCloseContest1 = ERC20Mock(weth).
                balanceOf(conMan);
27          uint256 playerBalanceAfterCloseContest1 = ERC20Mock(weth).
                balanceOf(player1);
28
29          uint256 managerCut1 = conManBalanceAfterCloseContest1 -
                conManBalanceBeforeCloseContest1;
30          console.log("managerCut1 ", managerCut1.toString());
31
32          uint256 playerProfit1 = playerBalanceAfterCloseContest1 -
                playerBalanceBeforeCloseContest1;
33          console.log("playerProfit1 ", playerProfit1.toString());
34
35          // close contest second time
36          uint256 conManBalanceBeforeCloseContest2 = ERC20Mock(weth).
                balanceOf(conMan);
37          uint256 playerBalanceBeforeCloseContest2 = ERC20Mock(weth).
                balanceOf(player1);
38          vm.prank(user);
39          ContestManager(conMan).closeContest(contest);
40          uint256 comManBalanceAfterCloseContest2 = ERC20Mock(weth).
                balanceOf(conMan);
41          uint256 playerBalanceAfterCloseContest2 = ERC20Mock(weth).
                balanceOf(player1);
42
43          uint256 managerCut2 = comManBalanceAfterCloseContest2 -
                conManBalanceBeforeCloseContest2;
44          console.log("managerCut2 ", managerCut2.toString());
45
46          uint256 playerProfit2 = playerBalanceAfterCloseContest2 -
                playerBalanceBeforeCloseContest2;
47          console.log("playerProfit2 ", playerProfit2.toString());
48
49          assertLt(managerCut1, managerCut1 + managerCut2);
50          assertLt(playerProfit1, playerProfit1 + playerProfit2);
51
52          // second player claimed
53          uint256 player2BalanceBeforeClaim = ERC20Mock(weth).balanceOf(
                player2);
54          vm.prank(player2);
55          Pot(contest).claimCut();
56          uint256 player2BalanceAfterClaim = ERC20Mock(weth).balanceOf(
                player2);
57          uint256 player2ProfitAfterClaim = player2BalanceAfterClaim -
                player2BalanceBeforeClaim;
58          console.log("player2ProfitAfterClaim ", player2ProfitAfterClaim
                .toString());
59          assertGt(player2ProfitAfterClaim, 0);
60      }
```

The logs show the actual doubled profit for a manager and a player after 2 consequent closings:

```
  managerCut1  150
  playerProfit1  337
  managerCut2  150
  playerProfit2  337
  player2ProfitAfterClaim  500
```

Recommended Mitigation: The `remainingRewards` value should be assigned to zero once a contest is closed. Follow check-effect-interact pattern:

```
function closePot() external onlyOwner {
        if (block.timestamp - i_deployedAt < 90 days) {
            revert Pot__StillOpenForClaim();
        }
-       if (remainingRewards > 0) {
+       if (remainingRewards == 0) {
+           return;
+       }
        uint256 managerCut = remainingRewards / managerCutPercent;
        uint256 claimantCut = (remainingRewards - managerCut) /
           i_players.length;
+       remainingRewards = 0;

        i_token.transfer(msg.sender, managerCut);

        for (uint256 i = 0; i < claimants.length; i++) {
            _transferReward(claimants[i], claimantCut);
        }
    }
}
```

[H-3] Tokens are trapped in `ContestManager.sol` forever because no function allow to withdraw them

- *Impact: High*
- *Likelihood: High*

Root + Impact

Description

- When a contest is over, a user calls `ContestManager::closeContest` to close a Pot. This function calls `Pot::closePot()` where `managerCut` is calculated and trasfered back to the `ContestManager.sol`. There is no way to withdraw tokens from `ContestManager.sol`.

Here are functions from `ContestManager.sol`:

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

function _closeContest(address contest) internal {
    Pot pot = Pot(contest);
@>    pot.closePot();
}
```

Here is `Pot::closePot()` function, where tokens are transferred to msg.sender which is a `ContestManager.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);

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

Risk

Likelihood:

- The issue occurs every time a contest is closed by an owner.

Impact:

- Transferred tokens to `ContestManager.sol` cannot be withdrawn and are trapped in the protocol forever.

Proof of Concept:

There is no way to withdraw funds from `ContestManager.sol`.

Recommended Mitigation Add a `withdraw` function to `ContestManager.sol`:

```
+event FundsWithdrawn(address indexed to, uint256 indexed amount);

+function withdraw() public onlyOwner {
+        token.safeTransfer(msg.sender, token.balanceOf(address(this)))
    ;
```

```
5 +         emit FundsWithdrawn(msg.sender, token.balanceOf(address(this))
      );
6 +     }
```

[H-4] Claimants receive less rewards than they deserve due to incorrect `claimantCut` calculation in `Pot::closePot` and `Pot.sol` holds the difference

- *Impact: High*
- *Likelihood: High*

Root + Impact

Description:

- It is stated in a protocol documentation that when a contest is closed and a manager takes his cut, 'the remainder is distributed equally to those who claimed in time!' but it is not distributed fairly.
- The remainder is divided by an amount of all players, not claimant, and this means that an eligible claimant will receive less than he should.
- The pot balance holds climantCuts for players that didn't claim.

```
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  @>          uint256 claimantCut = (remainingRewards - managerCut) /
       i_players.length;
10         for (uint256 i = 0; i < claimants.length; i++) {
11             _transferReward(claimants[i], claimantCut);
12         }
13     }
```

Risk

Likelihood:

- The issue occurs every time the contest is closed and the remainder of a pool is distributed among claimants.

Impact:

- Eligible claimants receive less tokens than they deserve.

Proof of Concept: Please, add the following test to `TestMyCut.sol`:

```
function test_claimantReceiveLess_potHoldsDifference() public
    mintAndApproveTokens {
        vm.startPrank(user);
        rewards = [500, 500];
        totalRewards = 1000;
        contest = ContestManager(conMan).createContest(players, rewards
            , IERC20(ERC20Mock(weth)), totalRewards);

        uint256 potBalBefore = ERC20Mock(weth).balanceOf(contest);
        console.log("potBalBefore ", potBalBefore.toString());

        ContestManager(conMan).fundContest(0);
        vm.stopPrank();

        // only one claimant
        vm.startPrank(player1);
        Pot(contest).claimCut();
        vm.stopPrank();

        vm.warp(91 days);

        vm.startPrank(user);
        ContestManager(conMan).closeContest(contest);
        vm.stopPrank();

        uint256 remainingRewards = Pot(contest).getRemainingRewards();
        uint256 managerPercent = 10;
        uint256 amountOfClaimers = 1;
        uint256 managerCut = remainingRewards / managerPercent;
        uint256 userClaimCutExpected = (remainingRewards - managerCut)
            / amountOfClaimers;
        uint256 userClaimCutActual = (remainingRewards - managerCut) /
            players.length;

        assertGt(userClaimCutExpected, userClaimCutActual);

        uint256 potBalAfter = ERC20Mock(weth).balanceOf(contest);
        console.log("potBalAfter ", potBalAfter.toString());

        assertEq(potBalBefore, 0);
        assertEq(potBalAfter, userClaimCutActual);
    }
```

Recommended Mitigation: Consider to divide remaining pool by the amount of claimants, not all players:

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

```
4       }
5       if (remainingRewards > 0) {
6           uint256 managerCut = remainingRewards / managerCutPercent;
7           i_token.transfer(msg.sender, managerCut);
8
9   -           uint256 claimantCut = (remainingRewards - managerCut) /
        i_players.length;
10  +           uint256 claimantCut = (remainingRewards - managerCut) /
        claimants.length;
11
12          for (uint256 i = 0; i < claimants.length; i++) {
13              _transferReward(claimants[i], claimantCut);
14          }
15      }
```

[H-5] Tokens are trapped in `Pot.sol` forever because no function allow to withdraw them

- *Impact: High*
- *Likelihood: High*

Root + Impact:

Description:

- `Pot.sol` contract can be funded several times and that donations are not accounted in `remainingRewards`, so tokens just stay in a contract.
- Along with this, `Pot.sol` balance increases now when `Pot::closePot()` is called - there is incorrect calculation of claimantCut, which leads to that some tokens stay in `Pot.sol`

Here is `Pot::closePot()` function, where tokens are transferred to msg.sender which is a `ContestManager.sol`:

```
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   @>              uint256 claimantCut = (remainingRewards - managerCut) /
        i_players.length;
10              for (uint256 i = 0; i < claimants.length; i++) {
11                  _transferReward(claimants[i], claimantCut);
12              }
13          }
```

Risk

Likelihood:

- The issue occurs every time a contest is closed by an owner or the Pot is funded severul times via `ContestManager::fundContest()` function.

Impact:

- Tokens in `Pot.sol` cannot be withdrawn and are trapped in the protocol forever.

Proof of Concept:

There is no way to withdraw funds from `Pot.sol`. Tokens stay in `Pot.sol` after a contest is closed. Here is a test that proves that `Pot.sol` balance is not zero when contest is closed:

```
function test_tokensStayInPot_whenContestIsClosed() public mintAndApproveTokens {
        vm.startPrank(user);
        rewards = [500, 500];
        totalRewards = 1000;
        contest = ContestManager(conMan).createContest(players, rewards, IERC20(ERC20Mock(weth)), totalRewards);

        uint256 potBalBefore = ERC20Mock(weth).balanceOf(contest);
        console.log("potBalBefore ", potBalBefore.toString());

        ContestManager(conMan).fundContest(0); // + 1000
        vm.stopPrank();

        uint256 potBalBeforeClaim = ERC20Mock(weth).balanceOf(contest);
        console.log("potBalBeforeClaim ", potBalBeforeClaim.toString());

        vm.startPrank(player1);
        Pot(contest).claimCut(); // -500
        vm.stopPrank();

        uint256 potBalAfterClaim = ERC20Mock(weth).balanceOf(contest);
        console.log("potBalAfterClaim ", potBalAfterClaim.toString());

        vm.prank(user);
        ContestManager(conMan).fundContest(0); // + 1000

        uint256 potBalAfterFund2 = ERC20Mock(weth).balanceOf(contest);
        console.log("potBalAfterFund2 ", potBalAfterFund2.toString());

        vm.warp(91 days);

        vm.startPrank(user);
```

```
32          ContestManager(conMan).closeContest(contest); // (500 - 50) / 2
                = 225, i.e. -225 as claimantCut
33          vm.stopPrank();
34
35          uint256 expectedPotBalanceAfter = 1225;
36          uint256 potBalAfter = ERC20Mock(weth).balanceOf(contest);
37          console.log("potBalAfter ", potBalAfter.toString());
38
39          assertEq(potBalBefore, 0);
40          assertEq(expectedPotBalanceAfter, potBalAfter);
41          assertGt(potBalAfter, potBalBefore);
42      }
```

Recommended Mitigation 1. Add a `withdraw` function to `Pot.sol` and withdraw tokens to `ContestManager.sol`:

```
1  +event FundsWithdrawn(address indexed to, uint256 indexed amount);
2
3  +function withdraw() public onlyOwner {
4  +         token.safeTransfer(msg.sender, token.balanceOf(address(this)))
       ;
5  +         emit FundsWithdrawn(msg.sender, token.balanceOf(address(this))
       );
6  +     }
```

2. Also, it may be acceptable to account additionally transferred funds in `Pot::remainingRewards` and fix incorrect `claimantCut` calculation:

```
1  -uint256 private remainingRewards;
2
3  constructor(address[] memory players, uint256[] memory rewards, IERC20
       token, uint256 totalRewards) {
4      i_players = players;
5      i_rewards = rewards;
6      i_token = token;
7      i_totalRewards = totalRewards;
8  -    remainingRewards = totalRewards;
9      i_deployedAt = block.timestamp;
10
11     // i_token.transfer(address(this), i_totalRewards);
12     for (uint256 i = 0; i < i_players.length; i++) {
13         playersToRewards[i_players[i]] = i_rewards[i];
14     }
15 }
16
17 function closePot() external onlyOwner {
18     if (block.timestamp - i_deployedAt < 90 days) {
19         revert Pot__StillOpenForClaim();
20     }
21 +   uint256 remainingRewards = i_token.balanceOf(address(this));
```

```
22      if (remainingRewards > 0) {
23          uint256 managerCut = remainingRewards / managerCutPercent;
24          i_token.transfer(msg.sender, managerCut);
25  -       uint256 claimantCut = (remainingRewards - managerCut) /
        i_players.length;
26  +       uint256 claimantCut = (remainingRewards - managerCut) /
        claimants.length;
27          for (uint256 i = 0; i < claimants.length; i++) {
28              _transferReward(claimants[i], claimantCut);
29          }
30      }
31  }
```

[H-6] Possible discrepancy between sum of all rewards and `totalRewards` causes incorrect accounting and may lead to underflow error

- *Impact: High*
- *Likelihood: High*

Root + Impact

Description

- To create a contest, user should pass an array of rewards for every player `uint256[] memory rewards` and the total rewards amount `uint256 totalRewards`. The total rewards amount is assigned to `remainingRewards` in the constructor of `Pot.sol` and this amount is decremented on every player's claim. If the sum of rewards from an array is not the same as `totalRewards` amount, the incorrect accounting and possible underflow will be when `Pot::claimCut()` and `Pot::closePot()` are called.

```
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          // i_token.transfer(address(this), i_totalRewards);
10
11         for (uint256 i = 0; i < i_players.length; i++) {
12             playersToRewards[i_players[i]] = i_rewards[i];
13         }
14     }
15
```

```
16  function claimCut() public {
17          address player = msg.sender;
18          uint256 reward = playersToRewards[player];
19          if (reward <= 0) {
20              revert Pot__RewardNotFound();
21          }
22          playersToRewards[player] = 0;
23  @>        remainingRewards -= reward;
24          claimants.push(player);
25          _transferReward(player, reward);
26      }
```

Risk:

Likelihood:

- The issue occurs when the sum of rewards is not equal to `totalRewards` parameter in `ContestManager::createContest()` function.

Impact:

- Depending of whether `totalRewards` is less or more than the sum of rewards in the array `uint256[] memory rewards`, the following scenarios may develop (remember that `totalRewards` value is assigned to `remainingRewards` in a Pot):
- – if (`totalRewards` < sum of rewards) - possible underflow in `Pot::claimCut()` when `remainingRewards -= reward;;`
- – if (`totalRewards` > sum of rewards) - the remaining amount will be more than it should be, then wrong 10% of manager's cut and wrong claimant cut will be sent to participants.

Proof of Concept: Please, add the following test to `TestMyCut.t.sol` to see that the issue causes: 1. panic: arithmetic underflow or overflow when `totalRewards` < sum of rewards 2. wrong calculating of `remainingRewards` and `managersCut` when `totalRewards` > sum of rewards

```
1   function test_totalRewardAndRewardsSumDiscrepancy() public
       mintAndApproveTokens {
2           // contest with totalRewards < sum of rewards
3           vm.startPrank(user);
4           rewards = [500, 500];
5           totalRewards = 950;
6           contest = ContestManager(conMan).createContest(players, rewards
               , IERC20(ERC20Mock(weth)), totalRewards);
7           ContestManager(conMan).fundContest(0);
8           vm.stopPrank();
9
10          vm.startPrank(player1);
11          Pot(contest).claimCut();
12          vm.stopPrank();
```

```
13
14          vm.startPrank(player2);
15          vm.expectRevert(stdError.arithmeticError); // reverts with '
                panic: arithmetic underflow or overflow'
16          Pot(contest).claimCut();
17          vm.stopPrank();
18
19          // contest2 with totalRewards > sum of rewards
20          vm.startPrank(user);
21          rewards = [500, 500];
22          totalRewards = 1950;
23          address contest2 = ContestManager(conMan).createContest(players
                , rewards, IERC20(ERC20Mock(weth)), totalRewards);
24          ContestManager(conMan).fundContest(1);
25          vm.stopPrank();
26
27          vm.startPrank(player1);
28          Pot(contest2).claimCut();
29          vm.stopPrank();
30
31          uint256 expectedRemainingRewards = 500;
32          uint256 actualRemainingRewards = Pot(contest2).
                getRemainingRewards();
33          assertGt(actualRemainingRewards, expectedRemainingRewards);
34
35          vm.warp(91 days);
36
37          // contest manager balance before close contest
38          uint256 conManBalanceBeforeCloseContest = ERC20Mock(weth).
                balanceOf(conMan);
39
40          vm.prank(user);
41          ContestManager(conMan).closeContest(contest2);
42
43          // contest manager balance after close contest
44          uint256 conManBalanceAfterCloseContest = ERC20Mock(weth).
                balanceOf(conMan);
45
46          uint256 managerCutPercent = 10;
47          uint256 expectedManagerCut = expectedRemainingRewards /
                managerCutPercent;
48          uint256 actualManagerCut = conManBalanceAfterCloseContest -
                conManBalanceBeforeCloseContest;
49          assertGt(actualManagerCut, expectedManagerCut);
50      }
```

Recommended Mitigation:

It will be more reliable to calculate `remainingRewards` in `Pot.sol` constructor based on values of rewards in `rewards` array rather than pass `totalRewards` to `ContestManager::`

`createContest()` function and `Pot.sol` constructor:

Consider these changes in `Pot.sol`:

```
- uint256 private immutable i_totalRewards;

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

        // i_token.transfer(address(this), i_totalRewards);

        for (uint256 i = 0; i < i_players.length; i++) {
            playersToRewards[i_players[i]] = i_rewards[i];
+           remainingRewards += i_rewards[i];
        }
    }
```

Consider these changes in `ContestManager::createContest()`:

```
-function createContest(address[] memory players, uint256[] memory
    rewards, IERC20 token, uint256 totalRewards)
+function createContest(address[] memory players, uint256[] memory
    rewards, IERC20 token)
        public
        onlyOwner
        returns (address)
    {
-        Pot pot = new Pot(players, rewards, token, totalRewards);
+        Pot pot = new Pot(players, rewards, token);

        contests.push(address(pot));
-       contestToTotalRewards[address(pot)] = totalRewards;
+       contestToTotalRewards[address(pot)] = pot.getRemainingRewards()
        return address(pot);
    }
```

[H-7] No protocol profit and it may be economically impractical to close a contest

- *Impact: High*
- *Likelihood: High*

Root + Impact

Description:

- By design, a `ContestManager.sol` contract receives 10% reward from pot's remaining rewards after the owner closed a contest. Before that during 90 days the participants are allowed to claim their reward. A contract will receive nothing if all participants decided to claim their reward or if the remaining reward is too small to cover even gas expenses.

```
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;
            for (uint256 i = 0; i < claimants.length; i++) {
                _transferReward(claimants[i], claimantCut);
            }
        }
}
```

Risk

Likelihood:

- The issue occurs when all players claimed their rewards before the contest is closed or when the manager's cut, which is 10% from remaining rewards, is too low to cover even gas expenses.

Impact:

- The purpose of the protocol is to have profit for players and an owner. The current design allows financial loss or absence of any profit for an owner.

Proof of Concept: Pleade, add the following tests to `TestMyCut.t.sol`: 1. `test_economicallyImpractical` shows that when all players claimed their reward, there is nothing left for `managerCut`. 2. `test_economicallyImpracticalToCloseContest_whenRewardIsLessThanGasUsed` shows that `managerCut` is too small even to cover gas usage for `closeContest()` transaction.

```
function
    test_economicallyImpracticalToCloseContest_whenAllPlayersClaimed()
    public mintAndApproveTokens {
        vm.startPrank(user);
        rewards = [500, 500];
```

```
4           totalRewards = 1000;
5           contest = ContestManager(conMan).createContest(players, rewards
                , IERC20(ERC20Mock(weth)), totalRewards);
6           ContestManager(conMan).fundContest(0);
7           vm.stopPrank();
8
9           // all players claimed
10          uint256 playersAmount = players.length;
11          for(uint256 i = 0; i < playersAmount; i++){
12              vm.startPrank(players[i]);
13              Pot(contest).claimCut();
14              vm.stopPrank();
15          }
16          uint256 conManBalanceBeforeCloseContest = ERC20Mock(weth).
                balanceOf(conMan);
17          console.log("conManBalanceBeforeCloseContest ",
                conManBalanceBeforeCloseContest.toString());
18          vm.warp(91 days);
19
20          vm.startPrank(user);
21          ContestManager(conMan).closeContest(contest);
22          vm.stopPrank();
23
24          uint256 conManBalanceAfterCloseContest = ERC20Mock(weth).
                balanceOf(conMan);
25          console.log("conManBalanceAfterCloseContest ",
                conManBalanceAfterCloseContest.toString());
26
27          assertEq(conManBalanceAfterCloseContest -
                conManBalanceBeforeCloseContest, 0);
28      }
29
30      function
            test_economicallyImpracticalToCloseContest_whenRewardIsLessThanGasUsed
            () public mintAndApproveTokens {
31          vm.startPrank(user);
32          rewards = [500, 500];
33          totalRewards = 1000;
34          contest = ContestManager(conMan).createContest(players, rewards
                , IERC20(ERC20Mock(weth)), totalRewards);
35          ContestManager(conMan).fundContest(0);
36          vm.stopPrank();
37
38          // one player claimed
39          vm.startPrank(player1);
40          Pot(contest).claimCut(); // one reward is left -> is 10% is
                more than  gas used?
41          vm.stopPrank();
42
43          vm.warp(91 days);
44
```

```
45          uint256 conManBalanceBeforeCloseContest = ERC20Mock(weth).
               balanceOf(conMan);
46
47          //gas
48          uint256 gasStart = gasleft();
49          vm.txGasPrice(960000000);//0.96 gwei per unit of gas
50          vm.prank(user);
51          ContestManager(conMan).closeContest(contest);
52          uint256 gasEnd = gasleft();
53          uint256 gasUsed = (gasStart - gasEnd) * tx.gasprice;
54          console.log("gas units used ", (gasStart - gasEnd).toString());
55          console.log("gasUsed ", gasUsed.toString());
56
57          uint256 comManBalanceAfterCloseContest = ERC20Mock(weth).
               balanceOf(conMan);
58
59          uint256 managerCut = comManBalanceAfterCloseContest -
               conManBalanceBeforeCloseContest;
60          console.log("managerCut ", managerCut.toString());
61
62          assertGt(gasUsed, managerCut);
63      }
```

Recommended Mitigation:

1. It may be acceptable to allow not all players to claim, so at least 1 reward of one player stays in the protocol.
2. Specify the minimum reward to cover at least gas expenses by 10% from it (at least 0.0005 ether)

```
1  + uint256 public constant MINIMUM_REWARD_AMOUNT = 0.0005 ether;
2  + error Pot__RewardIsLessThanMininumAllowed();
3  + error Pot__CannotClaimTheLastReward();
4
5  constructor(address[] memory players, uint256[] memory rewards, IERC20
       token, uint256 totalRewards) {
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          // i_token.transfer(address(this), i_totalRewards);
14
15          for (uint256 i = 0; i < i_players.length; i++) {
16  +           require(i_rewards[i] >= MINIMUM_REWARD_AMOUNT,
       Pot__RewardIsLessThanMininumAllowed());
17              playersToRewards[i_players[i]] = i_rewards[i];
18          }
19      }
```

```
20
21 function claimCut() public {
22         address player = msg.sender;
23         uint256 reward = playersToRewards[player];
24         if (reward <= 0) {
25             revert Pot__RewardNotFound();
26         }
27         playersToRewards[player] = 0;
28 +        require((remainingRewards - reward) > 0,
       Pot__CannotClaimTheLastReward());
29         remainingRewards -= reward;
30         claimants.push(player);
31         _transferReward(player, reward);
32     }
```

[M-1] Unbound array of players may cause `OutOfGas` error and DoS attack

- *Impact: High*
- *Likelihood: Low*

Root + Impact

Description

- To create a contest, user calls `ContestManager::createContest()` function, which accepts an array of players of arbitraty length and passes it to a `Pot.sol` constructor, where **`for`** `loop` iterates over players. If there are more than 14430 members in `address[] memory players`, the `OutOfGas` error occurs, contest creation will revert and a lot of gas will be spent.

Here is `ContestManager::createContest()` function:

```
1  @>     function createContest(address[] memory players, uint256[] memory
       rewards, IERC20 token, uint256 totalRewards)
2              public
3              onlyOwner
4              returns (address)
5          {
6              // Create a new Pot contract
7              Pot pot = new Pot(players, rewards, token, totalRewards);
8              contests.push(address(pot));
9              contestToTotalRewards[address(pot)] = totalRewards;
10             return address(pot);
11         }
```

Here is a `Pot.sol` constructor:

```
constructor(address[] memory players, uint256[] memory rewards, IERC20 token, uint256 totalRewards) {
        i_players = players;
        i_rewards = rewards;
        i_token = token;
        i_totalRewards = totalRewards;
        remainingRewards = totalRewards;
        i_deployedAt = block.timestamp;

        // i_token.transfer(address(this), i_totalRewards);

@>        for (uint256 i = 0; i < i_players.length; i++) {
            playersToRewards[i_players[i]] = i_rewards[i];
        }
    }
```

Risk

Likelihood:

- The issue occurs when more than 14430 players are sent to `ContestManager::createContest()` function.

Impact:

- If the user does not know about this limitation and attempts to create a contest, the contest will not be created and the gas will be spent.

Proof of Concept: Please add the following fuzz test to the `TestMyCut.t.sol`. The max amount of `totalRewards` is type(uint96).max since it is big enough for any balance. The minimum amount of players, when the function `ContestManager::createContest()` does not revert is 14429, so the test uses 14430 players to show the revert.

```
function test_unboundPlayersAmount_cause_OutOfGas_and_memoryOOG_fuzz(uint256 totalRewardsFuzz, uint256 playersAmount) public {
        uint256 playersAmountWhenOutOfGaz = 14430;

        totalRewardsFuzz = bound(totalRewardsFuzz, playersAmountWhenOutOfGaz, type(uint96).max);
        playersAmount = bound(playersAmount, 0, playersAmountWhenOutOfGaz); //zero players is tested, 1 wei reward is tested

        // arrange owner
        uint256 currentUserBalance = ERC20Mock(address(weth)).balanceOf(user);
        uint256 wethTotalSupply = weth.totalSupply();

        //to prevent overflow on mint consider totalSupply as well
```

```
12          if(currentUserBalance + wethTotalSupply < totalRewardsFuzz){
13              ERC20Mock(address(weth)).mint(user, totalRewardsFuzz -
                    currentUserBalance - wethTotalSupply);
14          }
15
16          weth.approve(address(conMan), totalRewardsFuzz);
17
18          uint256 oneReward = playersAmount == 0 ? totalRewardsFuzz :
                totalRewardsFuzz / playersAmount;
19          uint256 totalRewardsRounded = playersAmount == 0 ?
                totalRewardsFuzz : oneReward * playersAmount;
20
21          address[] memory playersLocal = new address[](playersAmount);
22          uint256[] memory rewardsLocal = new uint256[](playersAmount);
23
24           for(uint256 i = 0; i < playersAmount; i++){
25              address player = vm.randomAddress();
26              playersLocal[i] = player;
27              rewardsLocal[i] = oneReward;
28           }
29          vm.prank(user);
30          ContestManager(conMan).createContest(playersLocal, rewardsLocal
                , IERC20(ERC20Mock(weth)), totalRewardsRounded);
31      }
```

The output of the test above is the following:

```
1           -- [987656084] - new
                Pot@0x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0
2                   -- emit OwnershipTransferred(previousOwner: 0
                        x0000000000000000000000000000000000000000,
                        newOwner: ContestManager: [0
                        x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353])
3   @>              -- - [OutOfGas] 0
        x6080604052348015610000f575f80fd5b506004361061086575f3560e01c8063715018a61161005
```

Recommended Mitigation: To avoid this issue it is recommended: 1. Add the maximum amount of players as a constant. 2. Add a custom error to notify about the issue. 3. Check a length of `players` array when a user attempts to create a new contest.

```
1   + error ContestManager__MaximumAmountOfPlayersExceeded();
2   + uint256 public constant MAXIMUM_AMOUNT_OF_PLAYERS = 1429;
3
4   function createContest(address[] memory players, uint256[] memory
        rewards, IERC20 token, uint256 totalRewards)
5           public
6           onlyOwner
7           returns (address)
8       {
```

```
+         require(players.length < MAXIMUM_AMOUNT_OF_PLAYERS,
    ContestManager__MaximumAmountOfPlayersExceeded());
          // Create a new Pot contract
          Pot pot = new Pot(players, rewards, token, totalRewards);
          contests.push(address(pot));
          contestToTotalRewards[address(pot)] = totalRewards;
          return address(pot);
      }
```

Also consider that it may be too gas intensive to create a contes and expenses on gas will not be covered by manager's cut, which may be even zero.

[M-2] Possible inconsistency in `players` and `rewards` arrays length prevents contest creation

- *Impact: Medium*
- *Likelihood: Medium*

Root + Impact

Description:

- It is possible to pass arrays `address[] memory players` and `uint256[] memory rewards` of different length as params to `ContestManager::createContest()`, where the lengths are never validated.
- Later these arrays are passed to a `Pot.sol` constructor, where for each player from `address[] memory players` array a corresponding reward is assigned from a `uint256[] memory rewards` array and stored in a mapping `mapping(address => uint256)private playersToRewards`.

Here is `ContestManager::createContest()`:

```
@> function createContest(address[] memory players, uint256[] memory
    rewards, IERC20 token, uint256 totalRewards)
          public
          onlyOwner
          returns (address)
      {
          // Create a new Pot contract
          Pot pot = new Pot(players, rewards, token, totalRewards);
          contests.push(address(pot));
          contestToTotalRewards[address(pot)] = totalRewards;
          return address(pot);
      }
```

Here is `Pot.sol` constructor:

```
constructor(address[] memory players, uint256[] memory rewards, IERC20
    token, uint256 totalRewards) {
        i_players = players;
        i_rewards = rewards;
        i_token = token;
        i_totalRewards = totalRewards;
        remainingRewards = totalRewards;
        i_deployedAt = block.timestamp;

        // i_token.transfer(address(this), i_totalRewards);

        for (uint256 i = 0; i < i_players.length; i++) {
@>            playersToRewards[i_players[i]] = i_rewards[i];
        }
    }
```

Risk

Likelihood:

- The issue will occur when amount of members in `players` array is more than an amount of members in `rewards` array in params of `ContestManager::createContest`

Impact:

- It will not be possible to create a Pot, the transaction will revert but gas for the previous actions in `createContest()` function and `Pot.sol` constructor will be vasted.

Proof of Concept: Please, add the following test `test_revertsWhen_playersAndRewardsArraysLengthIs` to `TestMyCut.t.sol`. The amount of players is less than amount of rewards in arrays - players.length > rewards.length.

```
address player3 = makeAddr("player3");
address player4 = makeAddr("player4");
address[] playersLocal = [player1, player2, player3, player4];

function test_revertsWhen_playersAndRewardsArraysLengthIsDifferent()
    public mintAndApproveTokens {
        uint256 playersLocalLength = playersLocal.length;
        rewards = [500, 500, 500];
        uint256 rewardsLength = rewards.length;
        console.log("playersLocalLength ", playersLocalLength.toString
            ()); // 4
        console.log("rewardsLength ", rewardsLength.toString()); // 3

        assertLt(rewardsLength, playersLocalLength);
        totalRewards = 2500;

        vm.prank(user);
```

```
16          vm.expectRevert(stdError.indexOOBError);// fail with 'panic:
                array out-of-bounds access'
17          contest = ContestManager(conMan).createContest(playersLocal,
                rewards, IERC20(ERC20Mock(weth)), totalRewards);
18      }
```

Recommended Mitigation: Check length of both arrays and revert if the lengths are not equal:

```
1  +error ContestManager__PlayersAmountIsNotEqualRewardsAmount();
2
3  function createContest(address[] memory players, uint256[] memory
       rewards, IERC20 token, uint256 totalRewards)
4          public
5          onlyOwner
6          returns (address)
7      {
8  +        require(players.length == rewards.length,
       ContestManager__PlayersAmountIsNotEqualRewardsAmount());
9          // Create a new Pot contract
10         Pot pot = new Pot(players, rewards, token, totalRewards);
11         contests.push(address(pot));
12         contestToTotalRewards[address(pot)] = totalRewards;
13         return address(pot);
14     }
```

[L-1] The return value of `ERC20:transfer()` and `ERC20:transferFrom()` functions in never evaluated which may lead to silent fail of token transfer.

- *Impact: Low*
- *Likelihood: Low*

Root + Impact

Description:

- Some ERC20 tokens return **`false`** on failed transaction and do not revert. The return bool value of transfer() function is not checked and this may lead to wrong assumption that tokens are transferred when transfer() or transferFrom() returns false.

The issue is in the following code lines: 1. `Pot::closePot()`

```
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               uint256 claimantCut = (remainingRewards - managerCut) /
                    i_players.length;
10              for (uint256 i = 0; i < claimants.length; i++) {
11                  _transferReward(claimants[i], claimantCut);
12              }
13          }
14      }
```

2. `Pot::_transferReward()`

```
1  function _transferReward(address player, uint256 reward) internal {
2  @>          i_token.transfer(player, reward);
3      }
```

3. `ContestManager::fundContest()`:

```
1  function fundContest(uint256 index) public onlyOwner {
2          Pot pot = Pot(contests[index]);
3          IERC20 token = pot.getToken();
4          uint256 totalRewards = contestToTotalRewards[address(pot)];
5
6          if (token.balanceOf(msg.sender) < totalRewards) {
7              revert ContestManager__InsufficientFunds();
8          }
9
10 @>         token.transferFrom(msg.sender, address(pot), totalRewards);
11     }
```

Risk

Likelihood:

- The issue occurs when there is an attempt to transfer a token that returns **`false`** when a transfer failed.

Impact: * The protocol and user may compose a false assumption that transfer is successful, but it is not.

Proof of Concept: Some tokens do not return revert of transfer failure - ERC-20:no revert on failure

Recommended Mitigation Use SafeERC20 from OpenZeppelin and the transaction will revert if the tranfer is not successful:

```
1  + using SafeERC20 for IERC20;
2
3  function closePot() external onlyOwner {
4          if (block.timestamp - i_deployedAt < 90 days) {
5              revert Pot__StillOpenForClaim();
```

```
6           }
7           if (remainingRewards > 0) {
8               uint256 managerCut = remainingRewards / managerCutPercent;
9   -           i_token.transfer(msg.sender, managerCut);
10  +           i_token.safeTransfer(msg.sender, managerCut);
11              uint256 claimantCut = (remainingRewards - managerCut) /
                    i_players.length;
12              for (uint256 i = 0; i < claimants.length; i++) {
13                  _transferReward(claimants[i], claimantCut);
14              }
15          }
16      }
17
18  function _transferReward(address player, uint256 reward) internal {
19  -        i_token.transfer(player, reward);
20  +        i_token.safeTransfer(player, reward);
21      }
22
23  function fundContest(uint256 index) public onlyOwner {
24          Pot pot = Pot(contests[index]);
25          IERC20 token = pot.getToken();
26          uint256 totalRewards = contestToTotalRewards[address(pot)];
27
28          if (token.balanceOf(msg.sender) < totalRewards) {
29              revert ContestManager__InsufficientFunds();
30          }
31
32  -        token.transferFrom(msg.sender, address(pot), totalRewards);
33  +        token.safeTransferFrom(msg.sender, address(pot), totalRewards)
        ;
34      }
```