Protocol Audit Report

Version 1.0

Cyfrin.io

September 2, 2024

MyCut Audit Report

0xNascosta

September 2, 2024

Prepared by: Cyfrin Lead Auditors: - 0xNascosta

Table of Contents

Disclaimer

0xNascosta 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. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	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

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

```
946231db0fe717039429a11706717be568d03b54
```

Scope

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

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.

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	5
Medium	2
Low	2
Info	3
Gas	1

Findings

High

[H-1] `closePot` contains division that may lead to precision loss

Description: ERC20 tokens have a `decimals` value associated with them, in the case of WETH - the value is 18. Hence if we have an amount say:

`uint256 amount = 1 ether //1000000000000000000`, this would be 1 wETH. `uint256 amount = 0.001 ether //1000000000000000`, this would be 0.001 wETH.

When calculating the `claimantCut`, suppose we had a numerator of `1 ether` and 3 players to divide them amongst (we should use `claimants.length` but that's rasied in another issue). We would get a loss in precision:

```
claimantCut = 1000000000000000000 / 3 = 333333333333333333 units
```

In this case the divison would result in 0.333333333333333333 wETH to each claimant. There would be a remainder of 1 unit that is effectively lost in the division.

Impact: If we were distributing a large sum among many claimants, the cumulative loss could be come significant. Resulting in a loss of funds being sent to the claimants as well as the managers cut.

Proof of Concept: See `testRemainingRewardsAreIncorrectlyDistributed` within the ProofOfCodes within the audit directory which highlights a loss in precision.

Recommended Mitigation

Mitigation strategies include:

- Using a multiplier to preserve precision.
- Distributing the remainder to minimize the impact of truncation.
- SafeMath (in older Solidity versions) to prevent arithmetic errors.
- Avoiding operations on very small units (like wei) directly, to reduce precision loss.

[H-2] `fundContest` allows closed Pots to be funded

Description: The `fundContest` function does not check if a particular Pot / Contest has been closed.

```
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();
        }
        // audit: q - Assumes the msg.sender has approved the
           ContestManager to spend funds.
        token.transferFrom(msg.sender, address(pot), totalRewards);
    }
```

This can result in funds being sent to a closed pot if the `index` parameter used corresponds to an contest `address` that has been closed.

Impact: Funds sent to a closed pot will not be redeemable again by contestants, but only extractable by existing claimaints and the manager, when the manager decides to close the pot again. This could occur out of mistake given we fund via `index` and not an `address` directly. Fat finger error could lead to the incorrect or a closed contest being funded.

Proof of Concept:

Add the following function to the `TestMyCut.t.sol:TestMyCut` test contract suite.

```
    /**
        Demonstrating a closed pot being funded by the contest manager,
            resulting in unclaimable funds by the users, however funds
           can only be claimed when the owner re-closes the pot.
     */
    function testCanFundClosedContestAndClaimReverts() public
       mintAndApproveTokens {
        // Arrange
        vm.startPrank(user); // Owner of contest
        ContestManager contestManager = ContestManager(conMan);
        uint256 userInitialBalance = weth.balanceOf(user);
        console.log("User initial balance %d", userInitialBalance);
```

```
        contest = contestManager.createContest(players, rewards, weth,
            totalRewards);
        contestManager.fundContest(0);
        vm.stopPrank();
        // Let users claim
        Pot contestPot = Pot(contest);
        for (uint256 i = 0; i < players.length; i++) {
            vm.prank(players[i]);
            contestPot.claimCut(); // Cut has been claimed by users.
        }
        vm.warp(91 days);
        vm.prank(conMan);
        contestPot.closePot();

        uint256 playerOneInitialBalance = weth.balanceOf(player1);
        uint256 playerTwoInitialBalance = weth.balanceOf(player2);

        // Act - Here we can fund the closedPot contest again and users
            cannot claim.
        vm.prank(user);
        contestManager.fundContest(0); // We shouldn't allow funding a
            closed pot.
        for (uint256 i = 0; i < players.length; i++) {
            vm.expectRevert(Pot.Pot__RewardNotFound.selector);
            vm.prank(players[i]);
            contestPot.claimCut(); // Will revert as reward for each
                user is 0 as they have claimed.
        }

        // Assert
        uint256 playerOneSecondClaimBalance = weth.balanceOf(player1);
        uint256 playerTwoSecondClaimBalance = weth.balanceOf(player2);

        // Owner has funded contest twice, however users could only
            successfully claim once via claimCut.
        assert(playerOneInitialBalance == playerOneSecondClaimBalance);
        assert(playerTwoInitialBalance == playerTwoSecondClaimBalance);
        assert(userInitialBalance - weth.balanceOf(user) == 2 *
            totalRewards);
    }
```

Recommended Mitigation: `fundContest` checks the `contestToTotalRewards` mapping for the `totalRewards` value, however if this value was zero, then the attempt to fund the contest will revert. This suggests we should ensure that when we invoke `closeContest` we also remove the `contestToTotalRewards` entry for the contest address we are closing.

```
function _closeContest(address contest) internal {
+       delete contestToTotalRewards[contest];
        Pot pot = Pot(contest);
        pot.closePot();
```

```
5 }
```

[H-3] No validation on `players` and `rewards` within `ContestManager:createContest` or `Pot:constructor`

Description: By not validating the `players` and `rewards` array to contain appropriate values, i.e. forbidding the zero-address, ensuring non-zero values for the reward in addition to ensuring the lengths of the arrays are the same.

The `Pot:constructor` below, specifically the for-loop will reveral this bug:

```
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          // This line is commented, is this supposed to be uncommented?
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     }
```

Impact: In the event that `players.length > rewards.length` we will get a reversion in the `Pot:constructor` as the rewards index will become out of bounds. The pot will not create which is a reasonable reversion but it should be more explicit. In the event that `players.length < rewards.length`, unclaimable funds the sum of funds at `index > players.length` within `rewards` will not be claimable by any user but will have been sent to the contract, most likely these will be shared amongst claimants when the pot is closed.

Proof of Concept:

Add the following functions to the `TestMyCut.t.sol:TestMyCut` test contract suite.

```
1  /**
2       * If the number of users exceeds the length of the rewards array
           we expect a reversion.
3       */
4      function testRevertWhenLengthUsersGreaterThanRewards() public {
5          // Arrange - Length of (Players: 2, Rewards: 1)
6          rewards = [50];
7          totalRewards = 50;
8          vm.startPrank(user);
9          ContestManager contestManager = ContestManager(conMan);
10
11         // Act / Assert
```

```
12          vm.expectRevert();
13          contestManager.createContest(players, rewards, IERC20(ERC20Mock
                (weth)), totalRewards);
14      }
15
16      /**
17       * If the rewards array has a greater length than the players array
             , the rewards at index >= players.length
18       * will be unclaimable by any user until the pot is closed, it will
              be shared amongst claimants and the conMan.
19       */
20      function testFundsAreAllocatedToUnclaimableUser() public
            mintAndApproveTokens {
21          // Arrange
22          address[] memory contestants = new address[](2); // Length 2
23          contestants[0] = makeAddr("PlayerOne");
24          contestants[1] = makeAddr("PlayerTwo");
25          rewards = [5, 5, 10]; // Length 3 (The 10 WETH is not allocated
                to a specific user)
26          totalRewards = 20;
27
28          vm.startPrank(user);
29          ContestManager contestManager = ContestManager(conMan);
30          contest = contestManager.createContest(contestants, rewards,
                IERC20(ERC20Mock(weth)), totalRewards);
31          contestManager.fundContest(0);
32          Pot contestPot = Pot(contest);
33          vm.stopPrank();
34
35          // Act - All contestants a part of the contest have claimed,
                yet 10 ETH is still remaining.
36          for (uint256 i = 0; i < contestants.length; i++) {
37              vm.prank(contestants[i]);
38              contestPot.claimCut();
39          }
40
41          // Assert
42          uint256 remainingRewards = contestPot.getRemainingRewards();
43          assert(remainingRewards == rewards[rewards.length - 1]); //
                Unclaimable by any user until contest is closed.
44      }
```

Recommended Mitigation:

Ensure the lengths of both arrays are equal.

```
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   +       require(players.length == rewards.length, "Each player should
        have an associated reward.")
8           Pot pot = new Pot(players, rewards, token, totalRewards);
9           contests.push(address(pot));
10          contestToTotalRewards[address(pot)] = totalRewards;
11          return address(pot);
12      }
```

[H-4] No validation on `totalRewards` being equal to the sum of elements in `rewards`

Description: `totalRewards` is being manually passed in as a constructor parameter of `ContestManager`, there is no validation that `totalRewards == sum of the rewards array elements (rewardsSum)` There are two particular cases of interest:

- 1) `rewardsSum > totalRewards`: This scenario will lead to the Pot being under funded and unable to make payouts to users who attempt to claim their cut. There will be an arithmetic underflow when the protocol is unable to payout a particlar user within `claimCut`.

```
1  function claimCut() public {
2          // Follows CEI.
3          address player = msg.sender;
4          uint256 reward = playersToRewards[player];
5          if (reward <= 0) {
6              // audit: q - reward is a uint256 it's minimum value is 0,
                   it won't be less than 0.
7              revert Pot__RewardNotFound();
8          }
9          playersToRewards[player] = 0;
10         remainingRewards -= reward; // <-- Here we expect an underflow
11         claimants.push(player);
12         _transferReward(player, reward); // audit: q - This will revert
               is we have not funded the pot, perhaps a boolean state
              variable to check if funded or balance check.
13     }
```

- 2) `rewardsSum < totalRewards`: Users will be able to claim their cut, however the protocol will also be distributing additional funds to all claimants as there will be a surplus of the `ERC20` token used in the contest.

Impact: The two scenarios described above may result in users who cannot be paid out or users that have claimed their cut being paid an additional amount when the contest manager closes the pot.

Proof of Concept:

Add the below test functions to `TestMyCut.t.sol:TestMyCut` contract test suite.

```
    /**
     * Users will be able to claim their cut in addition to gaining a surplus amount of funds
     * when the contest is closed.
     */
    function testTotalRewardsGreaterThanRewardsElementsSum() public mintAndApproveTokens {
        // Arrange - Let both users claim their cut.
        totalRewards = 100;
        rewards = [40, 40];
        ContestManager contestManager = ContestManager(conMan);
        vm.startPrank(user);
        contest = contestManager.createContest(players, rewards, IERC20(ERC20Mock(weth)), totalRewards);
        contestManager.fundContest(0);
        vm.stopPrank();

        Pot contestPot = Pot(contest);
        for (uint256 i = 0; i < players.length; i++) {
            vm.prank(players[i]);
            contestPot.claimCut();
        }
        uint256 playerOnePostClaimBalance = weth.balanceOf(players[0]);
        uint256 playerTwoPostClaimBalance = weth.balanceOf(players[1]);

        // Act - Show the users receive additional funds when the pot is closed, despite having claimed already.
        vm.warp(91 days);
        vm.prank(user);
        contestManager.closeContest(contest);
        uint256 playerOneBalancePostPotClosure = weth.balanceOf(players[0]);
        uint256 playerTwoBalancePostPotClosure = weth.balanceOf(players[1]);
        assertTrue(playerOneBalancePostPotClosure > playerOnePostClaimBalance);
        assertTrue(playerTwoBalancePostPotClosure > playerTwoPostClaimBalance);
    }

    /**
     * Users may not be able to claim their cut.
     */
    function testTotalRewardsLessThanRewardsElementsSum() public mintAndApproveTokens {
        // Arrange
        totalRewards = 20;
        rewards = [20, 20];
        ContestManager contestManager = ContestManager(conMan);
        vm.startPrank(user);
        contest = contestManager.createContest(players, rewards, IERC20
```

```
            (ERC20Mock(weth)), totalRewards);
43          contestManager.fundContest(0);
44          vm.stopPrank();
45
46          // Act - Player 1 can claim but player 2 will be unable to
               claim their cut.
47          Pot contestPot = Pot(contest);
48          vm.prank(player1);
49          contestPot.claimCut();
50          // Assert - Expect reversion (Arithmetic underflow)
51          vm.expectRevert();
52          vm.prank(player2);
53          contestPot.claimCut();
54      }
```

Recommended Mitigation:

Although slightly more gas-inefficient, it is worth the additional gas then to have a mismatch between the sum of the element of the `rewards` array and the `totalRewards`.

```
1  - function createContest(address[] memory players, uint256[] memory
       rewards, IERC20 token, uint256 totalRewards)
2  + function createContest(address[] memory players, uint256[] memory
       rewards, IERC20 token)
3          public
4          onlyOwner
5          returns (address)
6      {
7  +       uint256 totalRewards;
8  +       for (uint256 i=0; i<rewards.length; i++) {
9  +         totalRewards += rewards[i];
10 +       }
11         // Create a new Pot contract
12         Pot pot = new Pot(players, rewards, token, totalRewards);
13         contests.push(address(pot));
14         contestToTotalRewards[address(pot)] = totalRewards;
15         return address(pot);
16     }
```

[H-5] Pot is not automatically closed after 90 days.

Description: The documentation states that authorised claimants (those addresses in the `players` array) can claim with 90 days, however the pot is not automatically closed in 90 days, it requires the `contestManager` to invoke `closeContest`. Unless the execution of `closeContest` is automated this will likely not occur on time, nor do we check the `block.timestamp` within `claimCut`. This will result in `players` still being able to claim after 90 days unless `closeContest` is invoked.

Impact: Authorised users can still claim beyond the 90 day limit, given that the `contestManager`

has not closed the contest. This reveals a mismatch in the behaviour of the protocol and the documentation.

Proof of Concept:

Add the following test function to `TestMyCut.t.sol:TestMyCut` test suite.

```
    /**
     * Users can still claim after 90 days have elapsed and the pot duration
     */
    function testUsersCanStillClaimCutAfter90Days() public mintAndApproveTokens {
        // Arrange
        vm.startPrank(user);
        ContestManager contestManager = ContestManager(conMan);
        contest = contestManager.createContest(players, rewards, IERC20(ERC20Mock(weth)), totalRewards);
        contestManager.fundContest(0);
        vm.stopPrank();
        Pot pot = Pot(contest);
        vm.warp(91 days);
        uint256 playerOneInitialBalance = weth.balanceOf(player1);

        // Act - User can still claim
        vm.prank(player1);
        pot.claimCut();
        uint256 playerOneFinalBalance = weth.balanceOf(player1);

        // Assert
        assertTrue(playerOneFinalBalance > playerOneInitialBalance);
    }
```

Recommended Mitigation:

There are 2 suggested approaches:

1) Utilise Chainlink Automation. Create a time-based trigger via Chainlink automation. `checkUpkeep` / `performUpkeep` should be implemented and the contract should implement `AutomationCompatibleInterface` - more details can be found on the Chainlink website referenced.

2) Add a `block.timestamp` check against the `i_deployedAt` time within `claimCut`.

```
+ error Pot__ContestDurationExceeded();

+ uint256 constant CONTEST_DURATION = 90 days;

  function claimCut() public {
+     if (block.timestamp - i_deployedAt > CONTEST_DURATION) {
```

```
+         revert Pot__ContestDurationExceeded();
+     }
        address player = msg.sender;
        uint256 reward = playersToRewards[player];
        if (reward <= 0) {
            revert Pot__RewardNotFound();
        }
        playersToRewards[player] = 0;
        remainingRewards -= reward;
        claimants.push(player);
        _transferReward(player, reward);
    }
```

Medium

[M-1] Remaining rewards in claimant cuts are not distributed correctly.

Description: As per the documentation *"the remainder is distributed equally to those who claimed in time"*, this is not true. The rewards are distributed equally, however not the *remainder* of the rewards. This is because the denominator we use to determine the claimant cut is `i_players.length` and it should actually be `claimants.length`.

Impact: By using `i_players.length` we are reducing the remaining rewards that the claimants should actually be receiving. As the claimants are a subset of players, the remaining rewards should only be split amongst them by the number of claimants not players.

Proof of Concept:

The below test can be added to the TestMyCut.t.sol:TestMyCut contract's test suite. It demonstrates the scenario whereby we have 2 players, each entitled to a reward of 50 WEI of wETH. A single user claims their cut $(50, \text{wWEI})$, and then the contest is closed. Once the manager takes their cut $\left(\frac{50,\text{wWEI}}{10} = 5, \text{wWEI}\right)$, the remaining reward $(45, \text{wWEI})$ should be sent to the claimant. However, it is divided by 2, highlighting a loss in precision. This results in only $22, \text{wWEI}$ being incorrectly sent to the claimant (with $1, \text{wWEI}$ lost due to precision loss), when the claimant is actually entitled to $45, \text{wWEI}$.

```
    event Transfer(address indexed from, address indexed to, uint256
        value);
    /**
     * Remaining rewards are incorrectly distributed.
     */
    function testRemainingRewardsAreIncorrectlyDistributed() public
        mintAndApproveTokens {
        // Arrange - A single user claims the reward and we close the
            contest.
        rewards = [50, 50];
```

```
        totalRewards = 100;
        ContestManager contestManager = ContestManager(conMan);
        vm.startPrank(user);
        contest = contestManager.createContest(players, rewards, IERC20(ERC20Mock(weth)), totalRewards);
        contestManager.fundContest(0);
        vm.stopPrank();

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

        uint256 contestManagerInitialBalance = weth.balanceOf(user);
        uint256 playerOneInitialBalance = weth.balanceOf(player1);
        uint256 playerTwoInitialBalance = weth.balanceOf(player2);

        // Act
        vm.warp(91 days);
        vm.prank(user);
        vm.expectEmit(true, true, false, true);
        emit Transfer(contest, player1, 22); // Highlighting a loss in precision
        contestManager.closeContest(contest);

        // Assert
        uint256 contestManagerFinalBalance = weth.balanceOf(user);
        uint256 playerOneFinalBalance = weth.balanceOf(player1);
        uint256 playerTwoFinalBalance = weth.balanceOf(player2);
        uint256 managersCut = 50 / 10;
        uint256 playerOneExpectedFinalBalance = rewards[0] + (50 - managersCut);
        assertFalse(playerOneFinalBalance == playerOneExpectedFinalBalance);
    }
```

Recommended Mitigation:

```
  function closePot() external onlyOwner {
          if (block.timestamp - i_deployedAt < 90 days) {
              // audit: q - magic number hard-coded.
              revert Pot__StillOpenForClaim();
          }
          if (remainingRewards > 0) {
              uint256 managerCut = remainingRewards / managerCutPercent; // audit: q - Division is throwing me off, SafeMath? This can go wrong leading to precision loss.
              i_token.transfer(msg.sender, managerCut);

-             uint256 claimantCut = (remainingRewards - managerCut) / i_players.length;
+             uint256 claimantCut = (remainingRewards - managerCut) /
```

```
        claimants.length;
12              for (uint256 i = 0; i < claimants.length; i++) {
13                  _transferReward(claimants[i], claimantCut);
14              }
15          }
16      }
```

[M-2] `remainingRewards` is not set to 0 after closing the pot. May result in additional funds being sent to claimants and the manager.

Description: The `closePot` function does not set `remainingRewards` to zero. Therefore if the `closePot` function were to be called my the `contestManager` the `remainingRewards` will be greater than 0 and execute the logic associated with determining the `managersCut` and splitting funds amongst `claimants` even though the logic had already been performed upon the initial closure of the pot.

```
1  function closePot() external onlyOwner {
2          if (block.timestamp - i_deployedAt < 90 days) {
3              // audit: q - magic number hard-coded.
4              revert Pot__StillOpenForClaim();
5          }
6          if (remainingRewards > 0) { // This needs to be set to 0 at the
               end of the function.
7              uint256 managerCut = remainingRewards / managerCutPercent;
                   // audit: q - Division is throwing me off, SafeMath?
                   This can go wrong leading to precision loss.
8              i_token.transfer(msg.sender, managerCut);
9
10             uint256 claimantCut = (remainingRewards - managerCut) /
                   i_players.length; // audit: q - We are splitting amongst
                    the claimaints, use the claimaints lenth not i_players.
11             for (uint256 i = 0; i < claimants.length; i++) {
12                 _transferReward(claimants[i], claimantCut);
13             }
14         }
15     }
```

Impact: Pot can be closed multiple times and an attempt at transferring tokens to the manager and the claimants will occur again, I believe this results in the protocol incorrectly closing the pot.

Proof of Concept:

The below test can be added to `TestMyCut.t.sol:TestMyCut` contracts test suite.

```
1      /**
2       * Demonstrating that the pot can be closed multiple times
3       * as remainingRewards was not set to 0.
4       */
5      function testRemainingRewardsNotSetToZeroPostPotClosure() public
```

```
        mintAndApproveTokens {
         // Arrange
         rewards = [500, 500];
         totalRewards = 1000;
         vm.startPrank(user);
         ContestManager contestManager = ContestManager(conMan);
         contest = contestManager.createContest(players, rewards, IERC20
            (ERC20Mock(weth)), totalRewards);
         contestManager.fundContest(0);
         vm.stopPrank();
         Pot pot = Pot(contest);

         vm.prank(players[0]);
         pot.claimCut();
         uint256 remainingFundsAfterFirstClaim = pot.getRemainingRewards
            ();

         // Act - We would expect the remaining funds to be sent to the
         vm.warp(91 days);
         vm.startPrank(user);
         contestManager.closeContest(contest);
         uint256 remainingFundsAfterPotClosed = pot.getRemainingRewards
            ();

         // Assert
         assertEq(remainingFundsAfterFirstClaim,
            remainingFundsAfterPotClosed);
         assertFalse(remainingFundsAfterPotClosed == 0);
     }
```

Recommended Mitigation:

```
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);
            }
         }
+        remainingRewards = 0;
    }
```

Low

[L-1] users can invoke `claimCut` prior to the contest being funded

Description: It is possible that once the contest has been created, it is not necessarily funded at the same time, these are separate operations, which may result in users attempting to invoke `claimCut`, however there would be no funds and we would most likely get a `ERC20InsufficientBalance` error. Users have most probably assumed that at the time of claiming their cut that the contest is funded. The more insidious issue lies in the fact that the timer of 90 days begins when the Pot contract is constructed not when it's funded, hence if the contract is not funded at the time of creation, users will not be entitled to the whole 90 day duration claim period.

Impact: Bad UX, as users would be able to attempt claim their cut but this would result in a reversion.

Proof of Concept:

The below test can be added to `TestMyCut.t.sol:TestMyCut` contracts test suite.

Recommended Mitigation:

We must ensure the contest is funded at the time it is created. Otherwise we should state a clearer error message.

In the event where we want to give the users a more gracious error message, we could add the following changes which leverages a boolean to track if the Pot has been funded:

```
contract Pot is Ownable(msg.sender) {
  /**
    Existing Code...
  */

+ boolean private s_isFunded; // Ensure this is updated correctly when the contract is funded.

  function claimCut() public {
+     if (!s_isFunded) {
+       revert Pot__InsufficientFunds();
+     }
      address player = msg.sender;
      uint256 reward = playersToRewards[player];
      if (reward <= 0) {
          revert Pot__RewardNotFound();
      }
      playersToRewards[player] = 0;
      remainingRewards -= reward;
      claimants.push(player);
      _transferReward(player, reward);
  }
}
```

In the scenario where we want to ensure the contest is funded at the time of being created employ the following code.

```
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;
+       fundContest(contests.length - 1);
        return address(pot);
    }

-   function fundContest(uint256 index) public onlyOwner {
+   function fundContest(uint256 index) internal 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);
    }
```

[L-2] `fundContest` index parameter is not validated.‘

Description: There is no validation on the fact that

Impact:

Proof of Concept:

Recommended Mitigation

Informational

[I-1] constants should be named in all capitals.

Description:

```
- uint256 private constant managerCutPercent = 10;
+ uint256 private constant MANAGER_CUT_PERCENT = 10;
```

[I-2] Magic numbers should be stored as variables.

Description:

`Pot.sol` uses 90 days as the duration of the contest, this should infact be extracted as a variable for improved readability.

```
+ uint256 public constant CONTEST_DURATION = 90 days;
```

[I-3] Contract has uses a floating version of the solidity.

Description: Contracts going in to production should use a fixed solidity version, the following should be added two both files:

```
- pragma solidity ^0.8.20;
+ pragma solidity 0.8.20;
```

Gas

[G-1] `immutable` keyword missing on `i_players` and `i_rewards`

Description: The `immutable` keyword is missing from the mentioned variables in the title.