MyCut Audit Report

Version 1.0

Owen Lee

September 16, 2024

MyCut Audit Report

Owen Lee

September 16, 2024

Prepared by: Owen Lee Lead Auditors: - Owen Lee.

Table of Contents

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

I, Owen Lee 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

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

Audit Details

- Commit Hash: None.

Scope

- In Scope:

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

Enjoyed the process of auditing this code, The developers of this codebase should spend a lot of time getting familiar with the common sets of vulnerabilities.

Issues found

Severity	Number of issues found
High	3
Medium	0
Low	3
Info	0
Gas Optimizations	0
Total	6

Findings

High

[H-1] Unbounded loop in `Pot::closePot` function that may lead to a DOS attack.

Description:

The `Pot` contract features an unbounded loop in the `closePot` function, which poses a significant risk of causing a Denial Of Service (DOS) attack.

```
1
2  function closePot() external onlyOwner {
3          if (block.timestamp - i_deployedAt < 90 days) {
4              revert Pot__StillOpenForClaim();
5          }
6          if (remainingRewards > 0) {
7              uint256 managerCut = remainingRewards / managerCutPercent;
8              i_token.transfer(msg.sender, managerCut);
9              uint256 claimantCut = (remainingRewards - managerCut) /
                   i_players.length;
10
11 @>             for (uint256 i = 0; i < claimants.length; i++) {
12                 _transferReward(claimants[i], claimantCut);
13             }
14         }
15     }
```

The unbounded nature of this loop introduces a risk where an unexpected surge of claimants could cause the contract to fail or execute critical functions.

Impact:

Function Execution Failure: In the `closePot` function, a large `claimants` array may result in the function failing to execute due to running out of gas. This failure could leave the pot open indefinitely, preventing the distribution of rewards and locking funds within the contract. This would disrupt the intended functionality and could lead to significant financial losses for users and the protocol.

Proof of Concept:

Unbounded Loop in closePot Function: - Simulate the scenario where a large number of claimants (e.g., 5,000 or more) have participated. - Call the `closePot` function. - Observe that the function fails to execute due to running out of gas, preventing the distribution of the remaining rewards.

Recommended Mitigation:

Redesign the reward distribution process to avoid large loops altogether. Consider implementing a

pull-based model where each `claimant` independently claims their reward without relying on the `closePot` function to distribute rewards.

[H-2] Unchecked duplicates in the `players` array may lead to an initialization error. (report submitted).

Description:

The `Pot` constructor initializes the `playersToRewards` mapping without checking for duplicate addresses in the `players` array. While this does not allow for multiple claims due to the `Pot::ClaimCut` function resetting rewards after a claim, it can still lead to incorrect initialization.

Impact:

Although duplicates do not lead to multiple reward claims, they can cause confusion and incorrect data in the initial setup. This could potentially mislead contract owners or users about the state of the rewards.

Proof of Concept:

1. Deploy the `Pot` contract with a `players` array that contains duplicates.
2. Observe that an duplicate address is initialized with the rewards for both instances in the array, but it can only claim once.

Add this to the `TestMyCut.t.sol` file.

This are the storage variables that have been used.

```
address duplicate1 = makeAddr("duplicate1");
address duplicate2 = makeAddr("duplicate1");

address[] duplicators = [duplicate1, duplicate2];
```

Then run this test function:

```
function testCanCreateDuplicatePlayersinAFundedcontest() public
    mintAndApproveTokens {
        console.log("This is the address of duplicate1::", duplicate1);
        console.log("This is the address of duplicate2::", duplicate2);

        // assertion passes
        assertEq(duplicate1, duplicate2);

        vm.startPrank(user);
        contest = ContestManager(conMan).createContest(duplicators,
            rewards, IERC20(ERC20Mock(weth)), 4);
        ContestManager(conMan).fundContest(0);
```

```
        vm.stopPrank();

        vm.startPrank(duplicate1);
        Pot(contest).claimCut();
        vm.stopPrank();

        vm.startPrank(duplicate2);
        vm.expectRevert();
        Pot(contest).claimCut();
        vm.stopPrank();


    }
```

These are the logs that were outputted:

```

Traces:
  [834406] TestMyCut::testCanCreateDuplicatePlayersinAFundedcontest()
    \- [0] console::log("Minting tokens to: ", user: [0
       x6CA6d1e2D5347Bfab1d91e883F1915560e09129D]) [staticcall]
    |   - -> [Stop]
    \- [0] VM::startPrank(user: [0
       x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
    |   - -> [Return]
    \- [29712] ERC20Mock::mint(user: [0
       x6CA6d1e2D5347Bfab1d91e883F1915560e09129D],
       100000000000000000000 [1e21])
    |   \- emit Transfer(from: 0
       x0000000000000000000000000000000000000000, to: user: [0
       x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], value:
       100000000000000000000 [1e21])
    |   - -> [Stop]
    \- [24739] ERC20Mock::approve(ContestManager: [0
       x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353],
       100000000000000000000 [1e21])
    |   \- emit Approval(owner: user: [0
       x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], spender:
       ContestManager: [0x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353],
       value: 100000000000000000000 [1e21])
    |   - -> [Return] true
    \- [0] console::log("Approved tokens to: ", ContestManager: [0
       x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353]) [staticcall]
    |   - -> [Stop]
    \- [0] VM::stopPrank()
    |   - -> [Return]
    \- [0] console::log("This is the address of duplicate1::",
       duplicate1: [0x729108660e7835278C57075d9a5050e6671c4d90]) [
       staticcall]
    |   - -> [Stop]
    \- [0] console::log("This is the address of duplicate2::",
```

```
            duplicate1: [0x729108660e7835278C57075d9a5050e6671c4d90]) [
            staticcall]
21      |   - -> [Stop]
22      \- [0] VM::assertEq(duplicate1: [0
            x729108660e7835278C57075d9a5050e6671c4d90], duplicate1: [0
            x729108660e7835278C57075d9a5050e6671c4d90]) [staticcall]
23      |   - -> [Return]
24      \- [0] VM::startPrank(user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
25      |   - -> [Return]
26      \- [612006] ContestManager::createContest([0
            x729108660e7835278C57075d9a5050e6671c4d90, 0
            x729108660e7835278C57075d9a5050e6671c4d90], [3, 1], ERC20Mock:
            [0x5929B14F2984bBE5309c2eC9E7819060C31c970f], 4)
27      |   \- [508191] -> new
            Pot@0x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0
28      |   |   \- emit OwnershipTransferred(previousOwner: 0
            x0000000000000000000000000000000000000000, newOwner:
            ContestManager: [0x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353])
29      |   |   - -> [Return] 1513 bytes of code
30      |   - -> [Return] Pot: [0x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0]
31      \- [29108] ContestManager::fundContest(0)
32      |   \- [192] Pot::getToken() [staticcall]
33      |   |   - -> [Return] ERC20Mock: [0
            x5929B14F2984bBE5309c2eC9E7819060C31c970f]
34      |   \- [641] ERC20Mock::balanceOf(user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D]) [staticcall]
35      |   |   - -> [Return] 100000000000000000000 [1e21]
36      |   \- [26084] ERC20Mock::transferFrom(user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], Pot: [0
            x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], 4)
37      |   |   \- emit Transfer(from: user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], to: Pot: [0
            x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], value: 4)
38      |   |   - -> [Return] true
39      |   - -> [Stop]
40      \- [0] VM::stopPrank()
41      |   - -> [Return]
42      \- [0] VM::startPrank(duplicate1: [0
            x729108660e7835278C57075d9a5050e6671c4d90])
43      |   - -> [Return]
44      \- [70849] Pot::claimCut()
45      |   \- [25188] ERC20Mock::transfer(duplicate1: [0
            x729108660e7835278C57075d9a5050e6671c4d90], 1)
46      |   |   \- emit Transfer(from: Pot: [0
            x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], to: duplicate1: [0
            x729108660e7835278C57075d9a5050e6671c4d90], value: 1)
47      |   |   - -> [Return] true
48      |   - -> [Stop]
49      \- [0] VM::stopPrank()
50      |   - -> [Return]
```

```
51      \- [0] VM::startPrank(duplicate1: [0
            x729108660e7835278C57075d9a5050e6671c4d90])
52      |   - -> [Return]
53      \- [0] VM::expectRevert(custom error f4844814:)
54      |   - -> [Return]
55      \- [440] Pot::claimCut()
56      |   - -> [Revert] Pot__RewardNotFound()
57      \- [0] VM::stopPrank()
58      |   - -> [Return]
59      - -> [Stop]
```

From the logs above, we can see that the 2nd time the duplicate address tried to use the `Pot::ClaimCut` function it reverted as you can tell by the output of the custom error(`Pot_RewardNotFound()`).

Recommended Mitigation:

Implement a check in the constructor to ensure that each address in the `Pot::players` array is unique. This can be done by adding the addresses to a `mapping` and reverting if a duplicate is found.

The following mitigation can be implemented in the `Pot` codebase as seen below..

```
1
2   +mapping(address => bool) private isPlayerAdded;
3
4       constructor(address[] memory players, uint256[] memory rewards,
            IERC20 token, uint256 totalRewards) {
5           i_players = players;
6           i_rewards = rewards;
7           i_token = token;
8           i_totalRewards = totalRewards;
9           remainingRewards = totalRewards;
10
11          i_deployedAt = block.timestamp;
12
13          for (uint256 i = 0; i < i_players.length; i++) {
14  +           if (isPlayerAdded[i_players[i]]) {
15  +               revert Pot__DuplicatePlayer();
16  +           }
17  +           isPlayerAdded[i_players[i]] = true;
18              playersToRewards[i_players[i]] = i_rewards[i];
19          }
20      }
```

Explanation:

1. **Duplicate Check:** A mapping called `isPlayerAdded` has been introduced. This mapping tracks whether a player has already been added to the `Pot::i_players` array.
2. **Constructor Update:** In the constructor, when players are added, the contract checks if they

are already present in the `isPlayerAdded` mapping. If a duplicate is found, the transaction is reverted with the custom error `Pot__DuplicatePlayer`.

3. **Claim and Close Functions:** The logic in `Pot::claimCut` and `Pot::closePot` remains unchanged, as this duplicate check ensures that the `i_players` array is unique from the beginning.

[H-3] Call inside loop in `Pot::closePot` function may address DOS.

Description:

The `Pot::closePot` function contains a loop that iterates over all `claimants` and processes their rewards distribution. Within this loop, there is a call to transfer funds to each `claimant`. Performing external calls inside a loop can be risky for several reasons:

1. **Gas Limit Exceeded:** If the number of claimants is large, the gas required to process all transfers could exceed the block gas limit, causing the transaction to fail.
2. **Denial of Service:** A malicious `claimant` could purposely cause the transfer to revert (e.g., by rejecting the transfer), which would revert the entire transaction, preventing other `claimants` from receiving their funds.

```
1
2  function closePot() external onlyOwner {
3          if (block.timestamp - i_deployedAt < 90 days) {
4              revert Pot__StillOpenForClaim();
5          }
6          if (remainingRewards > 0) {
7
8              uint256 managerCut = remainingRewards / managerCutPercent;
9
10             i_token.transfer(msg.sender, managerCut);
11             uint256 claimantCut = (remainingRewards - managerCut) /
                   i_players.length;
12
13
14             for (uint256 i = 0; i < claimants.length; i++) {
15 @>                _transferReward(claimants[i], claimantCut);
16             }
17         }
18     }
```

Impact:

The presence of a call inside the loop could lead to failed transactions, especially in scenarios with a large number of `claimants`. This could result in the pot remaining unclosed, funds being locked up, and `claimants` being unable to claim their cut.

Proof of Concept:

In a scenario where there are thousands of `claimants`, this function could exceed the gas limit and fail the transaction thus not sending any `claimant` cut to the thousands of `claimants`

Recommended Mitigation:

Use of Pull Payments: Instead of sending funds directly within the loop, consider implementing a pull payment mechanism where `claimants` can withdraw their rewards themselves. This approach minimizes the risks associated with external calls inside loops.

Low

[L-1] Unbounded loop in the `Pot::constructor` that may lead to a DOS attack.

Description:

The `Pot` contract features an unbounded loop in its constructor, which pose a significant risk of causing a Denial of Service (DOS) attack.

```
1
2  constructor(address[] memory players, uint256[] memory rewards, IERC20
       token, uint256 totalRewards) {
3          i_players = players;
4          i_rewards = rewards;
5          i_token = token;
6          i_totalRewards = totalRewards;
7          remainingRewards = totalRewards;
8          i_deployedAt = block.timestamp;
9
10 @>         for (uint256 i = 0; i < i_players.length; i++) {
11             playersToRewards[i_players[i]] = i_rewards[i];
12         }
13     }
```

The unbounded nature of these loop introduces a risk where an unexpected surge in players could cause the contract to fail to deploy or execute critical functions.

Impact:

Deployment Failure: If the `i_players` array in the constructor is excessively large, the contract's deployment will fail due to exceeding the block's gas limit. This failure effectively prevents the contract from being created and used.

Proof of Concept:

1. **Unbounded Loop in Constructor:**

- Attempt to deploy the `Pot` contract with a large `i_players` array (e.g., 10,000 or more players).
- The deployment will fail, and the constructor will not complete its execution due to exceeding the gas limit.

Recommended Mitigation:

Impose a maximum limit on the size of the `i_players` array. This limit ensures that the loop does not grow to a size that could exceed the block gas limit.

[L-2] Rounding error leading to potentially zero `manager cut`.

Description:

The `Pot::closePot` function calculates the `manager's` cut as a percentage of the remaining funds in the pot. However, due to Solidity's truncation behavior, the calculated cut could be rounded down to zero if the resulting amount is small enough. This could unfairly deprive the `manager` of their rightful cut, particularly in scenarios where the remaining funds are minimal.

```
1  function closePot() external onlyOwner {
2          if (block.timestamp - i_deployedAt < 90 days) {
3              revert Pot__StillOpenForClaim();
4          }
5          if (remainingRewards > 0) {
6
7  @>              uint256 managerCut = remainingRewards / managerCutPercent
       ;
8
9              i_token.transfer(msg.sender, managerCut);
10             uint256 claimantCut = (remainingRewards - managerCut) /
                   i_players.length;
11
12
13             for (uint256 i = 0; i < claimants.length; i++) {
14                 _transferReward(claimants[i], claimantCut);
15             }
16         }
17     }
```

Impact:

The `manager` could receive no compensation from the pot, undermining the financial incentives of managing the protocol. This is a critical issue as it directly affects the revenue model for `managers`.

Proof of Concept:

If the remaining funds in the pot are small, and the `manager's` cut (e.g., 10%) is calculated, the truncation during the division could lead to a result of zero, especially if the calculated cut is less than 1.

Add this to the `TestMyCut.t.sol` file.

```

function testManagerGetsUnfairZeroShare() public mintAndApproveTokens {
        vm.startPrank(user);
        // small rewards for each player which is 2 and 2
        rewards = [2, 2];
        // total reward which is 4
        totalRewards = 4;
        contest = ContestManager(conMan).createContest(players, rewards
            , IERC20(ERC20Mock(weth)), totalRewards);
        ContestManager(conMan).fundContest(0);
        vm.stopPrank();

        vm.startPrank(player1);
        Pot(contest).claimCut();
        vm.stopPrank();

        uint256 claimantBalanceBefore = ERC20Mock(weth).balanceOf(
            player1);
        // here the balance is 2

        vm.warp(91 days);

        vm.startPrank(user);
        ContestManager(conMan).closeContest(contest);
        vm.stopPrank();
        // Around here the managercut is 0 which is unfair.

        uint256 claimantBalanceAfter = ERC20Mock(weth).balanceOf(
            player1);
        // Here the player1 after the manager closes the contest his
            balance is 3

        console.log("This is the claimantBalanceBefore::",
            claimantBalanceBefore);
        console.log("This is the claimantBalanceAfter::",
            claimantBalanceAfter);
    }
```

These are the following logs that were Outputted::

```

Traces:
  [876492] TestMyCut::testManagerGetsUnfairZeroShare()
    \- [0] console::log("Minting tokens to: ", user: [0
        x6CA6d1e2D5347Bfab1d91e883F1915560e09129D]) [staticcall]
```

```
5       |   - -> [Stop]
6       \- [0] VM::startPrank(user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
7       |   - -> [Return]
8       \- [29712] ERC20Mock::mint(user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D],
            10000000000000000000 [1e21])
9       |   \- emit Transfer(from: 0
            x0000000000000000000000000000000000000000, to: user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], value:
            10000000000000000000 [1e21])
10      |   - -> [Stop]
11      \- [24739] ERC20Mock::approve(ContestManager: [0
            x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353],
            10000000000000000000 [1e21])
12      |   \- emit Approval(owner: user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], spender:
            ContestManager: [0x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353],
            value: 10000000000000000000 [1e21])
13      |   - -> [Return] true
14      \- [0] console::log("Approved tokens to: ", ContestManager: [0
            x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353]) [staticcall]
15      |   - -> [Stop]
16      \- [0] VM::stopPrank()
17      |   - -> [Return]
18      \- [0] VM::startPrank(user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
19      |   - -> [Return]
20      \- [634006] ContestManager::createContest([0
            x7026B763CBE7d4E72049EA67E89326432a50ef84, 0
            xEb0A3b7B96C1883858292F0039161abD287E3324], [2, 2], ERC20Mock:
            [0x5929B14F2984bBE5309c2eC9E7819060C31c970f], 4)
21      |   \- [530191] -> new
            Pot@0x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0
22      |   |   \- emit OwnershipTransferred(previousOwner: 0
            x0000000000000000000000000000000000000000, newOwner:
            ContestManager: [0x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353])
23      |   |   - -> [Return] 1513 bytes of code
24      |   - -> [Return] Pot: [0x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0]
25      \- [29108] ContestManager::fundContest(0)
26      |   \- [192] Pot::getToken() [staticcall]
27      |   |   - -> [Return] ERC20Mock: [0
            x5929B14F2984bBE5309c2eC9E7819060C31c970f]
28      |   \- [641] ERC20Mock::balanceOf(user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D]) [staticcall]
29      |   |   - -> [Return] 10000000000000000000 [1e21]
30      |   \- [26084] ERC20Mock::transferFrom(user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], Pot: [0
            x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], 4)
31      |   |   \- emit Transfer(from: user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], to: Pot: [0
```

```
            x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], value: 4)
32      |   |   - -> [Return] true
33      |   - -> [Stop]
34      \- [0] VM::stopPrank()
35      |   - -> [Return]
36      \- [0] VM::startPrank(player1: [0
            x7026B763CBE7d4E72049EA67E89326432a50ef84])
37      |   - -> [Return]
38      \- [70849] Pot::claimCut()
39      |   \- [25188] ERC20Mock::transfer(player1: [0
            x7026B763CBE7d4E72049EA67E89326432a50ef84], 2)
40      |   |   \- emit Transfer(from: Pot: [0
            x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], to: player1: [0
            x7026B763CBE7d4E72049EA67E89326432a50ef84], value: 2)
41      |   |   - -> [Return] true
42      |   - -> [Stop]
43      \- [0] VM::stopPrank()
44      |   - -> [Return]
45      \- [641] ERC20Mock::balanceOf(player1: [0
            x7026B763CBE7d4E72049EA67E89326432a50ef84]) [staticcall]
46      |   - -> [Return] 2
47      \- [0] VM::warp(7862400 [7.862e6])
48      |   - -> [Return]
49      \- [0] VM::startPrank(user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
50      |   - -> [Return]
51      \- [12074] ContestManager::closeContest(Pot: [0
            x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0])
52      |   \- [11219] Pot::closePot()
53      |   |   \- [5288] ERC20Mock::transfer(ContestManager: [0
            x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353], 0)
54      |   |   |   \- emit Transfer(from: Pot: [0
            x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], to: ContestManager:
            [0x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353], value: 0)
55      |   |   |   - -> [Return] true
56      |   |   \- [3288] ERC20Mock::transfer(player1: [0
            x7026B763CBE7d4E72049EA67E89326432a50ef84], 1)
57      |   |   |   \- emit Transfer(from: Pot: [0
            x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], to: player1: [0
            x7026B763CBE7d4E72049EA67E89326432a50ef84], value: 1)
58      |   |   |   - -> [Return] true
59      |   |   - -> [Stop]
60      |   - -> [Stop]
61      \- [0] VM::stopPrank()
62      |   - -> [Return]
63      \- [641] ERC20Mock::balanceOf(player1: [0
            x7026B763CBE7d4E72049EA67E89326432a50ef84]) [staticcall]
64      |   - -> [Return] 3
65      \- [0] console::log("This is the claimantBalanceBefore::", 2) [
            staticcall]
66      |   - -> [Stop]
```

```
67      \- [0] console::log("This is the claimantBalanceAfter::", 3) [
            staticcall]
68      |   - -> [Stop]
69      - -> [Stop]
```

As you can see where we have the trace for the `manager` cut::

```
1  - [12074] ContestManager::closeContest(Pot: [0
       x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0])
2       |   \- [11219] Pot::closePot()
3       |   |   \- [5288] ERC20Mock::transfer(ContestManager: [0
            x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353], 0)
4       |   |   |   \- emit Transfer(from: Pot: [0
            x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], to: ContestManager:
            [0x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353], value: 0)
5       |   |   |   - -> [Return] true
```

The `manager` unfairly gets a cut of 0 mainly because of truncation. This could be found where we have the manager cut formula::

```

uint256 managerCut = remainingRewards / managerCutPercent;
```

Here the remainingRewards for this scenario was `2`. So `2/10 = 0.5`. In solidity, there is no handling of fractional arithmetic so it gets truncate to `0`.

Recommended Mitigation:

To prevent truncation errors, consider implementing a minimum threshold for the `manager's` cut or using rounding-up techniques to ensure the `manager` always receives a non-zero amount when eligible.

[L-3] Rounding error in `claimant` Cut calculation.

Description:

Similarly, rounding errors can occur while calculating each claimant's cut, potentially leading to underpayment due to truncation. While this works against the favor of the user, it can still result in dissatisfaction and potential disputes if `claimants` feel they are not receiving their fair share.

```

function closePot() external onlyOwner {
        if (block.timestamp - i_deployedAt < 90 days) {
            revert Pot__StillOpenForClaim();
        }
        if (remainingRewards > 0) {

```

```
8               uint256 managerCut = remainingRewards / managerCutPercent;
9
10              i_token.transfer(msg.sender, managerCut);
11  @>            uint256 claimantCut = (remainingRewards - managerCut) /
        i_players.length;
12
13
14              for (uint256 i = 0; i < claimants.length; i++) {
15                  _transferReward(claimants[i], claimantCut);
16              }
17          }
18      }
```

Impact:

`claimants` may receive less than their entitled share of the remaining funds, which could lead to loss of trust in the protocol.

Proof of Concept:

If the remaining funds are not evenly divisible by the number of claimants, the truncation could result in under-distribution, leading some funds undistributed.

Add this to the `TestMyCut.t.sol` file.

```
1
2   function testClaimantGetsUnfairZeroShare() public mintAndApproveTokens
        {
3           vm.startPrank(user);
4           // small rewards for each player which is 2,2 and 2
5           rewards = [2, 2, 2];
6           // total reward which is 6
7           totalRewards = 6;
8           contest = ContestManager(conMan).createContest(players, rewards
                , IERC20(ERC20Mock(weth)), totalRewards);
9           ContestManager(conMan).fundContest(0);
10          vm.stopPrank();
11
12          vm.startPrank(player1);
13          Pot(contest).claimCut();
14          vm.stopPrank();
15
16          vm.startPrank(player2);
17          Pot(contest).claimCut();
18          vm.stopPrank();
19
20          vm.startPrank(player3);
21          Pot(contest).claimCut();
22          vm.stopPrank();
23
24
```

```
25          uint256 claimantBalanceBefore1 = ERC20Mock(weth).balanceOf(
                player1);
26          // here the balance is 2
27          uint256 claimantBalanceBefore2 = ERC20Mock(weth).balanceOf(
                player2);
28           // here the balance is 2
29           uint256 claimantBalanceBefore3 = ERC20Mock(weth).balanceOf(
                 player3);
30           // here the balance is 2
31
32          vm.warp(91 days);
33
34          vm.startPrank(user);
35          ContestManager(conMan).closeContest(contest);
36          vm.stopPrank();
37          // Around here the managercut is 0 which is unfair.
38
39          uint256 claimantBalanceAfter1 = ERC20Mock(weth).balanceOf(
                player1);
40          // Here the player1 after the manager closes the contest his
                balance is 2
41
42          uint256 claimantBalanceAfter2 = ERC20Mock(weth).balanceOf(
                player2);
43           // Here the player2 balance after the manager closes the
                 contest is 2
44
45          uint256 claimantBalanceAfter3 = ERC20Mock(weth).balanceOf(
                player3);
46          // Here the player3 balance after the manager closes the
                contest is 2
47
48          console.log("This is the claimantBalanceBefore1::",
                claimantBalanceBefore1);
49          console.log("This is the claimantBalanceBefore2::",
                claimantBalanceBefore2);
50          console.log("This is the claimantBalanceBefore3::",
                claimantBalanceBefore3);
51          console.log("This is the claimantBalanceAfter1::",
                claimantBalanceAfter1);
52          console.log("This is the claimantBalanceAfter2::",
                claimantBalanceAfter2);
53          console.log("This is the claimantBalanceAfter3::",
                claimantBalanceAfter3);
54      }
```

These are the following logs that were outputted::

```
1  Traces:
2    [1723902] TestMyCut::setUp()
3      \- [0] VM::startPrank(user: [0
```

```
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
4       |   - -> [Return]
5       \- [1103083] -> new
            ContestManager@0x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353
6       |   \- emit OwnershipTransferred(previousOwner: 0
            x0000000000000000000000000000000000000000, newOwner: user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
7       |   - -> [Return] 5391 bytes of code
8       \- [517943] -> new
            ERC20Mock@0x5929B14F2984bBE5309c2eC9E7819060C31c970f
9       |   \- emit Transfer(from: 0
            x0000000000000000000000000000000000000000, to: DefaultSender: [0
            x1804c8AB1F12E6bbf3894d4083f33e07309d1f38], value: 100000000000
            [1e11])
10      |   - -> [Return] 2124 bytes of code
11      \- [0] console::log("User Address: ", user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D]) [staticcall]
12      |   - -> [Stop]
13      \- [0] console::log("Contest Manager Address 1: ", ContestManager:
            [0x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353]) [staticcall]
14      |   - -> [Stop]
15      \- [0] VM::stopPrank()
16      |   - -> [Return]
17      - -> [Stop]
18
19    [1076746] TestMyCut::testClaimantGetsUnfairZeroShare()
20      \- [0] console::log("Minting tokens to: ", user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D]) [staticcall]
21      |   - -> [Stop]
22      \- [0] VM::startPrank(user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
23      |   - -> [Return]
24      \- [29712] ERC20Mock::mint(user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D],
            100000000000000000000 [1e21])
25      |   \- emit Transfer(from: 0
            x0000000000000000000000000000000000000000, to: user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], value:
            100000000000000000000 [1e21])
26      |   - -> [Stop]
27      \- [24739] ERC20Mock::approve(ContestManager: [0
            x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353],
            100000000000000000000 [1e21])
28      |   \- emit Approval(owner: user: [0
            x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], spender:
            ContestManager: [0x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353],
            value: 100000000000000000000 [1e21])
29      |   - -> [Return] true
30      \- [0] console::log("Approved tokens to: ", ContestManager: [0
            x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353]) [staticcall]
31      |   - -> [Stop]
```

```
32      \- [0] VM::stopPrank()
33      |   - -> [Return]
34      \- [0] VM::startPrank(user: [0
           x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
35      |   - -> [Return]
36      \- [702007] ContestManager::createContest([0
           x7026B763CBE7d4E72049EA67E89326432a50ef84, 0
           xEb0A3b7B96C1883858292F0039161abD287E3324, 0
           xcC37919fDb8E2949328cDB49E8bAcCb870d0c9f3], [2, 2, 2], ERC20Mock
           : [0x5929B14F2984bBE5309c2eC9E7819060C31c970f], 6)
37      |   \- [597792] -> new
           Pot@0x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0
38      |   |   \- emit OwnershipTransferred(previousOwner: 0
           x0000000000000000000000000000000000000000, newOwner:
           ContestManager: [0x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353])
39      |   |   - -> [Return] 1513 bytes of code
40      |   - -> [Return] Pot: [0x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0]
41      \- [29108] ContestManager::fundContest(0)
42      |   \- [192] Pot::getToken() [staticcall]
43      |   |   - -> [Return] ERC20Mock: [0
           x5929B14F2984bBE5309c2eC9E7819060C31c970f]
44      |   \- [641] ERC20Mock::balanceOf(user: [0
           x6CA6d1e2D5347Bfab1d91e883F1915560e09129D]) [staticcall]
45      |   |   - -> [Return] 100000000000000000000 [1e21]
46      |   \- [26084] ERC20Mock::transferFrom(user: [0
           x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], Pot: [0
           x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], 6)
47      |   |   \- emit Transfer(from: user: [0
           x6CA6d1e2D5347Bfab1d91e883F1915560e09129D], to: Pot: [0
           x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], value: 6)
48      |   |   - -> [Return] true
49      |   - -> [Stop]
50      \- [0] VM::stopPrank()
51      |   - -> [Return]
52      \- [0] VM::startPrank(player1: [0
           x7026B763CBE7d4E72049EA67E89326432a50ef84])
53      |   - -> [Return]
54      \- [70849] Pot::claimCut()
55      |   \- [25188] ERC20Mock::transfer(player1: [0
           x7026B763CBE7d4E72049EA67E89326432a50ef84], 2)
56      |   |   \- emit Transfer(from: Pot: [0
           x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], to: player1: [0
           x7026B763CBE7d4E72049EA67E89326432a50ef84], value: 2)
57      |   |   - -> [Return] true
58      |   - -> [Stop]
59      \- [0] VM::stopPrank()
60      |   - -> [Return]
61      \- [0] VM::startPrank(player2: [0
           xEb0A3b7B96C1883858292F0039161abD287E3324])
62      |   - -> [Return]
63      \- [48949] Pot::claimCut()
```

```
64      |   \- [25188] ERC20Mock::transfer(player2: [0
           xEb0A3b7B96C1883858292F0039161abD287E3324], 2)
65      |   |   \- emit Transfer(from: Pot: [0
           x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], to: player2: [0
           xEb0A3b7B96C1883858292F0039161abD287E3324], value: 2)
66      |   |   - -> [Return] true
67      |   - -> [Stop]
68      \- [0] VM::stopPrank()
69      |   - -> [Return]
70      \- [0] VM::startPrank(player3: [0
           xcC37919fDb8E2949328cDB49E8bAcCb870d0c9f3])
71      |   - -> [Return]
72      \- [48949] Pot::claimCut()
73      |   \- [25188] ERC20Mock::transfer(player3: [0
           xcC37919fDb8E2949328cDB49E8bAcCb870d0c9f3], 2)
74      |   |   \- emit Transfer(from: Pot: [0
           x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], to: player3: [0
           xcC37919fDb8E2949328cDB49E8bAcCb870d0c9f3], value: 2)
75      |   |   - -> [Return] true
76      |   - -> [Stop]
77      \- [0] VM::stopPrank()
78      |   - -> [Return]
79      \- [641] ERC20Mock::balanceOf(player1: [0
           x7026B763CBE7d4E72049EA67E89326432a50ef84]) [staticcall]
80      |   - -> [Return] 2
81      \- [641] ERC20Mock::balanceOf(player2: [0
           xEb0A3b7B96C1883858292F0039161abD287E3324]) [staticcall]
82      |   - -> [Return] 2
83      \- [641] ERC20Mock::balanceOf(player3: [0
           xcC37919fDb8E2949328cDB49E8bAcCb870d0c9f3]) [staticcall]
84      |   - -> [Return] 2
85      \- [0] VM::warp(7862400 [7.862e6])
86      |   - -> [Return]
87      \- [0] VM::startPrank(user: [0
           x6CA6d1e2D5347Bfab1d91e883F1915560e09129D])
88      |   - -> [Return]
89      \- [1425] ContestManager::closeContest(Pot: [0
           x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0])
90      |   \- [570] Pot::closePot()
91      |   |   - -> [Stop]
92      |   - -> [Stop]
93      \- [0] VM::stopPrank()
94      |   - -> [Return]
95      \- [641] ERC20Mock::balanceOf(player1: [0
           x7026B763CBE7d4E72049EA67E89326432a50ef84]) [staticcall]
96      |   - -> [Return] 2
97      \- [641] ERC20Mock::balanceOf(player2: [0
           xEb0A3b7B96C1883858292F0039161abD287E3324]) [staticcall]
98      |   - -> [Return] 2
99      \- [641] ERC20Mock::balanceOf(player3: [0
           xcC37919fDb8E2949328cDB49E8bAcCb870d0c9f3]) [staticcall]
```

```
100     |   - -> [Return] 2
101     \- [0] console::log("This is the claimantBalanceBefore1::", 2) [
            staticcall]
102     |   - -> [Stop]
103     \- [0] console::log("This is the claimantBalanceBefore2::", 2) [
            staticcall]
104     |   - -> [Stop]
105     \- [0] console::log("This is the claimantBalanceBefore3::", 2) [
            staticcall]
106     |   - -> [Stop]
107     \- [0] console::log("This is the claimantBalanceAfter1::", 2) [
            staticcall]
108     |   - -> [Stop]
109     \- [0] console::log("This is the claimantBalanceAfter2::", 2) [
            staticcall]
110     |   - -> [Stop]
111     \- [0] console::log("This is the claimantBalanceAfter3::", 2) [
            staticcall]
112     |   - -> [Stop]
113     - -> [Stop]
```

As you can see where we have the traces for the `claimant` cut::

```
1
2  - [641] ERC20Mock::balanceOf(player1: [0
       x7026B763CBE7d4E72049EA67E89326432a50ef84]) [staticcall]
3       |   - -> [Return] 2
4       \- [641] ERC20Mock::balanceOf(player2: [0
            xEb0A3b7B96C1883858292F0039161abD287E3324]) [staticcall]
5       |   - -> [Return] 2
6       \- [641] ERC20Mock::balanceOf(player3: [0
            xcC37919fDb8E2949328cDB49E8bAcCb870d0c9f3]) [staticcall]
7       |   - -> [Return] 2
```

From the above trace, it proves that none of the 3 `claimants` got any share mainly because of the truncation that occurs in the `claimant` cut formula below:

```
1
2  uint256 claimantCut = (remainingRewards - managerCut) / i_players.
       length;
```

From above, we all know that the manager cut gets truncated to 0.

So this how the formula works it out...

`remainingRewards` = 0 `managerCut` = 0 `i_players.length` = 3

`0 / 3` = 0

So the `claimantCut` ends up being 0.

Recommended Mitigation:

Rounding the final `claimantCut` division upwards would fix this.