MyCut Security Audit Report by MK (15th DEC 2024)

High Risk Findings

[H-1] Owner's cut is stuck in ContestManager

Description: The `closePot` function attempts to transfer the manager's cut directly to `msg.sender`, which is the `ContestManager` contract. Since this contract does not have a withdrawal function to handle ERC20 token transfers, the manager's cut tokens will remain locked in the `ContestManager` contract after the pot is closed.

Impact: Tokens intended as the manager's cut will be locked and inaccessible, since the `ContestManager` contract does not support ERC20 token withdrawals.

Proof of Concept:

1. The `closePot` function contains the following code to transfer the manager's cut:

```
i_token.transfer(msg.sender, managerCut);
```

2. Since each Pot instance is deployed by the `ContestManager`, the pot owner in this case will refers to the `ContestManager` contract. In this current implementation, the `ContestManager` contract does not have a mechanism to withdraw the ERC20 tokens, leaving the manager's cut tokens locked and inaccessible.

Recommended Mitigation:

1. Add a withdraw function to the `ContestManager` contract that allows the owner to claim ERC20 tokens. This function should be callable only by the contract owner and should be used to withdraw the manager's cut after each `closePot` action. This ensures that the manager's cut tokens are accessible and can be properly claimed.

[H-2] Incorrect reward calculation in Pot::closePot

Description: Reward calculation in Pot::closePot function does not follow its specs.

Impact:

1. Players has claimed will get less reward then they should
2. Players has not claimed will get the same reward as who has claimed (expected they get nothing)

Proof of Concept:

1. The specs for the `Pot::closePot` is as follow: "...the manager takes a cut of the remaining pool and the remainder is distributed equally to those who claimed **in time!** "

2. This mean unclaimed reward will be shared with manager and who has claimed.

3. But in the https://github.com/Cyfrin/2024-08-MyCut/blob/main/src/Pot.sol#L57, the remaining reward after manager take the cut, is shared to all players, includes who has not claimed yet.

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

Recommended Mitigation:

1. Divide the remaining reward to to the number of user who has claimed.

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

[H-3] Incorrect handling of duplicate addresses

Description: The `Pot` contract's constructor does not validate the uniqueness of player addresses when creating a new instance. This creates a risk where a player can appear multiple times in the `players` array, leading to incorrect reward allocation. In its current state, if duplicate player addresses are present, the reward is overwritten instead of incremented, which could cause distribution inconsistencies.

Impact: Duplicate addresses will either cause rewards to be overwritten or inconsistently distributed. This leads to an incorrect allocation of funds and undermines the fairness of the distribution mechanism.

Proof of Concept: The constructor of the `Pot` contract takes two arrays, `players` and `rewards`, to assign specific reward amounts to each player. However, no check is implemented to ensure that the `players` array does not contain duplicate addresses. When duplicate player addresses are present, only the last occurrence in the array will have its reward value assigned, as earlier values will be overwritten in the `playersToRewards` mapping.

Recommended Mitigation: Modify the logic to increment the reward for player addresses rather than overwriting it. This ensures that if a player appears multiple times in the `players` array, they receive the correct cumulative reward.

[H-4] Unbound loop in closePot resulting in DoS

Description: In certain conditions transaction will run out of gas and revert, causing DoS

Impact: The primary impact of this vulnerability is a Denial of Service (DoS) attack vector. An attacker (or even normal usage with a large number of claimants) can cause the `closePot` function to fail due to excessive gas consumption. This prevents the distribution of remaining rewards and the execution of any subsequent logic in the function, potentially locking funds in the contract indefinitely. In the case of smaller pots it would be a gas inefficency to itterate over the state variabel `claimants`.

Proof of Concept:

1. In Pot.sol, lines 58-60

2. If there are too many claimants, transaction will run out of gas and revert, causing DoS

```
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;
          // @audit bad practice! if there are too many claimants, the transaction
will run out of gas and revert
            for (uint256 i = 0; i < claimants.length; i++) {
                _transferReward(claimants[i], claimantCut);
            }
        }
```

Recommended Mitigation: Make another claim mechanism for remaining tokens, this time for claimants only.

Medium Risk Findings

[M-1] Incorrect Handling of Zero Claimants in `closePot()` Function

Description: In the `closePot` function, if the number of claimants is zero, the remaining rewards intended for distribution among claimants may not be properly reclaimed by the Contest Manager. The `claimantCut` is calculated using the length of the `i_players` array instead of the `claimants` array, which could lead to incorrect distribution. Additionally, the function does not have a mechanism to handle the scenario where there are zero claimants, resulting in the potential loss of rewards.

Impact:

1. This bug can lead to incomplete recovery of rewards by the Contest Manager. If no participants claim their rewards, a significant portion of the remaining tokens could remain locked in the contract indefinitely, leading to financial loss and inefficient fund management.
2. And All the reward is lost except from the little 10 % the manager gets because there was no mechanism to claim the remainingReward

Proof of Concept:

1. Add this test in the TestMyCut.t.sol:

```
function testClosePotWithZeroClaimants() public mintAndApproveTokens {
    vm.startPrank(user);

    // Step 1: Create a new contest
    contest = ContestManager(conMan).createContest(players, rewards, IERC20(weth),
totalRewards);
```

```
    // Step 2: Fund the pot
    ContestManager(conMan).fundContest(0);

    // Step 3: Move forward in time by 90 days so the pot can be closed
    vm.warp(block.timestamp + 90 days);

    // Step 4: Close the pot with 0 claimants
    uint256 managerBalanceBefore = weth.balanceOf(user);
    ContestManager(conMan).closeContest(contest);
    uint256 managerBalanceAfter = weth.balanceOf(user);

    vm.stopPrank();

    // Step 5: Assert that the Contest Manager received all the remaining rewards
    // Since there are no claimants, the manager should receive all remaining
rewards
    assertEq(managerBalanceAfter, managerBalanceBefore + totalRewards, "Manager
did not reclaim all rewards after closing pot with zero claimants.");
```

2. In the test `testClosePotWithZeroClaimants`, after closing a pot with zero claimants, the Contest Manager is unable to reclaim all the remaining rewards:

```
├─ [9811] ContestManager::closeContest(Pot:
[0x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0])
    │   ├─ [8956] Pot::closePot()
    │   │   ├─ [5288]
0x5929B14F2984bBE5309c2eC9E7819060C31c970f::transfer(ContestManager:
[0x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353], 0)
    │   │   │   ├─ emit Transfer(from: Pot:
[0x43e82d2718cA9eEF545A591dfbfD2035CD3eF9c0], to: ContestManager:
[0x7BD1119CEC127eeCDBa5DCA7d1Bd59986f6d7353], value: 0)
```

Recommended Mitigation:

1. Adjust Calculation Logic: Modify the `claimantCut` calculation to divide by `claimants.length` instead of `i_players.length`. This ensures that only the claimants are considered when distributing the remaining rewards.
2. Handle Zero Claimants: Implement a check to determine if there are zero claimants. If true, all remaining rewards should be transferred back to the Contest Manager to ensure no tokens are left stranded in the contract. Example:

```
  if (claimants.length == 0) {
      i_token.transfer(msg.sender, remainingRewards);
  } else {
      for (uint256 i = 0; i < claimants.length; i++) {
          _transferReward(claimants[i], claimantCut);
      }
  }
```

[M-2] Pot funded late results in <90 days to claim.

Description: Players have a 90 days window to claim reward which begins to count immediately after the contest is created. which is not fair to players because the contest is not funded in the same transaction as the creation.

Impact: Players could lose their rewards

Proof of Concept:

1. If the owner creates a contest and not fund it, the players will not be able to claim their rewards because it has not been funded, therefore the players are losing time until the contest is funded before they can claim, therefore the claim period starting immediately the contest is created whether funded or not is not fair to players.
2. The owner could decide to fund few seconds to when the 90 days is almost over, this way players might not be able to withdraw in time and the owner closes the pot and they get a big share of the reward when they close the p0t

Recommended Mitigation: the 90days countdown should start when the pot has been funded

Low Risk Findings

[L-1] Unbound for loop in Contest Creation

Description: In constructor of Pot contract, there is a for loop through `i_players` array and modify `playersToRewards` mapping (a state variable) inside it, those work will consume a lot of gas, and the longer of i_players array is, the more gas have to spend for this transaction.

Impact: If the loop consuming more gas than the block's gas limit, the transaction will revert.

Proof of Concept:

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

```
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;

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

```
        }
    }
```

2. 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.

Recommended Mitigation:

1. 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] Missing Events

Description The `ContestManager` contract lacks event emissions for critical actions such as creating, funding, and closing contests. Events are crucial in Solidity contracts for logging important actions, as they provide an immutable record on the blockchain that can be indexed and queried by off-chain applications. Without these events, tracking the state and history of the contract becomes difficult, which can hinder monitoring, debugging, and auditing efforts.

Impact

1. Without events, external parties (such as users or monitoring systems) cannot easily track when contests are created, funded, or closed. This lack of visibility can lead to difficulties in verifying the correct operation of the contract.
2. Developers and auditors will find it more challenging to diagnose issues or verify contract behavior without event logs. In the event of a bug or issue, the absence of events makes it harder to trace the sequence of actions leading up to the problem.
3. Participants in the contests might not have a clear understanding of the state of the contract, which could lead to mistrust or uncertainty.

Proof of Concept:

The following functions in the `ContestManager` contract are identified as lacking event emissions:

- `createContest`
- `fundContest`
- `closeContest`

For example, in the `createContest` function:

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

```
    // No event emitted to signal that a new contest has been created.
    return address(pot);
}
```

Recommended Mitigation: Add event declarations and emit statements in the contract to log significant actions.

```
event ContestCreated(address indexed contestAddress, uint256 totalRewards);
event ContestFunded(address indexed contestAddress, uint256 amount);
event ContestClosed(address indexed contestAddress);
```

[L-3] Precision loss can lead to rewards getting stuck in the pot forever

Description: When contest manager closes the pot by calling `Pot::closePot`, 10 percent of the remaining rewards are transferred to the contest manager and the rest are distributed equally among the claimants. It does this by dividing the rewards by the manager's cut percentage which is 10. Then the remaining rewards are divided by the number of players to distribute equally among claimants. Since solidity allows only integer division this will lead to precision loss which will cause a portion of funds to be left in the pot forever. Each pot follows the same method, so as number of pots grow, the loss of funds is very significant.

Impact: Reward tokens get stuck in the pot forever which causes loss of funds.

Proof of code:

1. Add the below test to `test/TestMyCut.t.sol`

```
function testPrecisionLoss() public mintAndApproveTokens {
        ContestManager cm = ContestManager(conMan);
        uint playersLength = 3;
        address[] memory p = new address[](playersLength);
        uint256[] memory r = new uint256[](playersLength);
        uint tr = 86;

        p[0] = makeAddr("_player1");
        p[1] = makeAddr("_player2");
        p[2] = makeAddr("_player3");
        r[0] = 20;
        r[1] = 23;
        r[2] = 43;

        vm.startPrank(user);
        address pot = cm.createContest(p, r, weth, tr);
        cm.fundContest(0);
        vm.stopPrank();

        console.log("\n\ntoken balance in pot before: ", weth.balanceOf(pot));
```

```
        vm.prank(p[1]); // player 2
        Pot(pot).claimCut();

        vm.prank(p[0]); // player 1
        Pot(pot).claimCut();

        vm.prank(user);
        vm.warp(block.timestamp + 90 days + 1);
        cm.closeContest(pot);

        console.log(
            "\n\ntoken balance in pot after closing pot: ",
            weth.balanceOf(pot)
        );

        assert(weth.balanceOf(pot) != 0);
    }
```

2. Run the below test command in terminal

```
forge test --mt testPrecisionLoss -vv
```

3. Which results in the below output

```
[⠒] Compiling...
[⠃] Compiling 1 files with 0.8.20
[⠊] Solc 0.8.20 finished in 2.57s
Compiler run successful!

Ran 1 test for test/TestMyCut.t.sol:TestMyCut
[PASS] testPrecisionLoss() (gas: 936926)
Logs:


token balance in pot before:  86


token balance in pot after closing pot:  1

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.75ms (654.60µs CPU
time)

Ran 1 test suite in 261.16ms (1.75ms CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)
```

4. If you observe the output you can see the pot still has rewards despite distributing them to claimants.

Recommended Mitigations:

1. Utilize a fixed-point arithmetic library or implement a custom solution to handle fee calculations with greater precision.