Mycut Competitive Audit Report

Version 1.0

Dec3mber

September 3, 2024

MyCut Competitive Audit Report

Dec3mber

September 3, 2024

Prepared by: Dec3mber Lead Auditors: - Dec3mber

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

The Dec3mber team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

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

Audit Details

- Commit Hash: 946231db0fe717039429a11706717be568d03b54

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

We spend 2 days using manual review, and successfully identified three vulnerabilities that pose potential risks to the security and functionality of the system.

Each of these vulnerabilities has been categorized based on its severity and potential impact on the overall security of the smart contract. Our detailed analysis and recommended mitigation strategies are provided in the following sections of this report. Addressing these issues promptly will be crucial to ensuring the safety and reliability of the smart contract.

Issues found

Severity	Numbers of issues found
High	3
Medium	4
Low	1
total	8

Findings

High

[H-01] Lack of check of `Pot::remainingRewards` at the end of `Pot::closePot`, can result in pot not being closed properly

Summary

The `Pot::closePot` may fail to fully distribute all remaining rewards. This can result in leftover funds within the contract after the pot is closed, potentially leading to unclaimed rewards, incorrect close pot and unexpected behavior.

Vulnerability Details

The `Pot::closePot` may fail to fully distribute all remaining rewards due to transfer failed or potential precision loss in integer division and incorrectly distribution of `Pot::remainingRewards` at

```
    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);
            }
        }
    }
```

This can result in leftover funds within the contract after the pot is closed, potentially leading to unclaimed rewards and unexpected behavior.

Impact

The `pot` is not close correctly, there may have funds in the contract, and if the funds is bigger than player's reward, then an unclaim player can claim cut after the `pot`is closed.

Tools Used

Manual Review

Recommendations

Transfer the rest of funds to the owner, and add a check in `Pot::closePot` function

```
    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;
+           if (address(this).balance > 0) {
+               i_token.safeTransfer(msg.sender, address(this).balance);
+           }
        }
    }
```

[H-02] Incorrect Calculation of Claimant's Share `Pot::claimantCut` in `Pot::closePot` Function

Summary

The `Pot::closePot` function incorrectly calculates the share of rewards to be distributed to claimants. Instead of dividing the remaining rewards by the number of actual claimants, it incorrectly uses the total number of players. This mistake could lead to an inaccurate distribution of rewards, where claimants receive less than they are entitled to.

Vulnerability Details

In the `Pot::closePot` function, the calculation of each claimant's share is incorrectly performed using the total number of players `Pot::i_players.length` rather than the actual number of claimants `Pot::claimants.length`:

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

Impact

- **Underpayment to Claimants**: The incorrect calculation can result in claimants receiving less than they are entitled to.

- **Residual Rewards**: The miscalculation could leave residual rewards in the contract, creating an inconsistency where not all funds are correctly distributed or claimed.

Tools Used

Manual Review

Recommendations

Correct the calculation in the `Pot::closePot` function to ensure that rewards are distributed based on the actual number of claimants:

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

This change ensures that only those who have actually claimed their rewards are considered in the distribution, leading to a fair and accurate allocation of the remaining rewards.

[H-03] Misallocation of `Pot::managerCut` to the `ContestManager` Contract Instead of Owner, locked the funds

Summary

The `Pot::closePot` function incorrectly transfers the `Pot::managerCut` to the `ContestManager` contract instead of the owner of the `ContestManager`. Additionally, the `ContestManager` contract lacks a function to allow the owner to withdraw these funds, leading to potential fund lockup.

Vulnerability Details

In the `Pot::closePot` function, when the pot is closed, the `Pot::managerCut`—a portion of the remaining rewards—is transferred to `msg.sender`:

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

Since `Pot::closePot` is marked with `onlyOwner`, the `msg.sender` here refers to the `ContestManager` contract itself, not the actual owner of the `ContestManager`. As a result, the `managerCut` is transferred to the `ContestManager` contract instead of the intended recipient, the owner of the `ContestManager`.

Furthermore, the `ContestManager` contract does not provide any function that allows the owner to withdraw these funds. This omission means that the funds transferred as `Pot::managerCut` could become permanently locked within the `ContestManager` contract, making them inaccessible to the intended recipient.

Impact

Locked Funds: The `Pot::managerCut` funds may become permanently locked within the `ContestManager` contract, rendering them inaccessible to the owner. This could result in a significant loss of funds, especially if the `Pot::managerCut` represents a substantial portion of the remaining rewards.

Tools Used

Manual Review

Recommendation

Consider implementing a `withdrawal` function in the `ContestManager` contract that allows the owner to withdraw any funds that may have been incorrectly transferred to the contract:

```
function withdrawTokens(IERC20 token, uint256 amount) external
    onlyOwner {
     token.transfer(msg.sender, amount);
}
```

Or update the `Pot::closePot` function to correctly transfer the `Pot::managerCut` directly to the owner of the `ContestManager` contract rather than to `msg.sender`.

Medium

[M-01] Unchecked transferFrom Operation in `ContestManager::fundContest`

Summary

The `ContestManager::fundContest` function uses an unchecked transferFrom operation, which may lead to situations where tokens are not properly transferred to the Pot contract without the contract detecting the failure.

Vulnerability Details

The`ContestManager::fundContest` attempts to transfer tokens from the caller to a specific Pot contract using the following code:

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

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

@>      token.transferFrom(msg.sender, address(pot), totalRewards);
    }
```

However, this operation does not check the return value of `ContestManager::token.transferFrom`. If the operation fails (e.g., due to insufficient balance or allowance), the contract will proceed as if the transfer was successful, potentially leading to a mismatch between the expected and actual funds available in the `Pot`.

Impact

Funds Mismatch: The `Pot` contract may not receive the intended amount of tokens, leading to insufficient rewards available for distribution. This could cause users to attempt to claim rewards that do not exist, resulting in failed transactions and a poor user experience.

Tools Used

Manual Review

Slither

Recommendations

Use OpenZeppelin's SafeERC20 library to ensure the transferFrom operation is checked for success, and revert the transaction if it fails.

```
    function fundContest(uint256 index) public onlyOwner {
        Pot pot = Pot(contests[index]); //@audit everytime declare pot
            contract, why use a storage variable to store?
        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);
+        token.safeTransferFrom(msg.sender, address(pot), totalRewards);
    }
```

Or check the return value.

```
function fundContest(uint256 index) public onlyOwner {
        Pot pot = Pot(contests[index]); //@audit everytime declare pot
            contract, why use a storage variable to store?
        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);
+        (bool success, ) = token.safeTransferFrom(msg.sender, address(
    pot), totalRewards);
+        if (!success) {
+            revert();
+        }
    }
```

[M-02] Unchecked transfer Operation in `Pot::closePot`

Summary

The `Pot::closePot` uses an unchecked transfer operation when transferring the manager's cut. This could lead to situations where the transfer fails without being detected, causing issues with the funds distribution.

Vulnerability Details

Within the `Pot::closePot`, the following code is used to transfer the manager's cut from the remaining rewards:

```
    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);
                }
            }
        }
```

This operation does not verify whether the transfer was successful. If it fails, the contract will continue executing, incorrectly assuming the manager has received their cut.

Impact

• **Manager's Funds**: The manager may not receive their entitled portion of the rewards.

• **Incorrect Distribution**: The remaining rewards, meant to be distributed among claimants, could be miscalculated if the manager's cut is not successfully transferred.

Tools Used

Manual Review

Slither

Recommendations

Replace the unchecked transfer with a safe transfer using OpenZeppelin's SafeERC20 library to ensure the operation's success:

```
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);
+       i_token.safeTransfer(msg.sender, managerCut);

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

[M-03] Unchecked transfer Operation in `Pot::_transferReward`

Summary

The `Pot::_transferReward`uses an unchecked transfer operation to distribute rewards to users. This could result in situations where the reward transfer fails without detection, potentially leaving users without their expected rewards.

Vulnerability Details

The _transferReward function handles the distribution of rewards to individual users:

```
    function _transferReward(address player, uint256 reward) internal {
        i_token.transfer(player, reward);
    }
```

This transfer operation does not check the return value to ensure the transfer was successful. If the transfer fails, the function will not revert, leading the contract to believe the transfer was successful when it was not.

Impact

- **User Rewards**: Users may not receive their rewards if the transfer fails.
- **Locked Funds**: If rewards are not transferred as expected, they may remain locked in the contract, potentially leading to funds being stuck and unusable.

Tools Used

Manual Review

Slither

Recommendations

Implement the SafeERC20 library's safeTransfer function to handle transfers and ensure the operation's success:

```
    function _transferReward(address player, uint256 reward) internal {
-       i_token.transfer(player, reward);
+       i_token.safeTransfer(player, reward);
    }
```

[M-04] `Pot::remainingRewards` Not Updated in `Pot::closePot` Function

Summary

The `Pot::remainingRewards` variable is not updated within the `Pot::closePot` function, leading to potential inconsistencies between the actual state of the contract and what is reported by the `Pot::getRemainingRewards` function. Whether `i_token.transfer(msg.sender, managerCut);` or `_transferReward(claimants[i], claimantCut);` transcation fails or not, you can't get the right result through the `Pot::getRemainingRewards` function.

Vulnerability Details

In the `Pot` contract, the `Pot::remainingRewards` variable tracks the amount of unclaimed rewards left in the pot. However, in the `Pot::closePot` function, where rewards are distributed to the manager and claimants, the `Pot::remainingRewards` is not updated after these transactions. This oversight can cause the `Pot::getRemainingRewards` function to return an outdated or incorrect value:

```
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);
@>      // Missing update to remainingRewards after giving manager cut
        uint256 claimantCut = (remainingRewards - managerCut) /
            i_players.length;
        for (uint256 i = 0; i < claimants.length; i++) {
            _transferReward(claimants[i], claimantCut);
        }

@>      // Missing update to remainingRewards after distribution
    }
}
```

Without updating `Pot::remainingRewards` to reflect the actual distribution, the value returned by `Pot::getRemainingRewards` may incorrectly indicate that there are still funds available in the pot, even after it has been closed and the funds have been distributed.

Impact

- **Incorrect Reward Reporting**: Users may be misled by incorrect values returned by `Pot::getRemainingRewards`, believing there are still rewards to be claimed when in fact the pot has already been closed and funds distributed.
- **Inconsistent Contract State**: The contract state may become inconsistent, leading to potential issues in future transactions or audits, as the actual remaining rewards do not match what is reported.
- **Potential Mismanagement**: Inaccurate reporting of remaining rewards can lead to the mismanagement of funds, where further actions are taken based on incorrect data

Tools Used

Manual Review

Recommendations

To resolve this issue, ensure that `Pot::remainingRewards` is correctly updated after distributing rewards in the `Pot::closePot` function:

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

+       remainingRewards = 0; // Ensure remainingRewards is updated
    after all distributions
    }
}
```

This change ensures that `Pot::remainingRewards` accurately reflects the current state of the contract after the `pot` has been closed and all funds have been distributed.

Low

[L-01] Incorrect Initialization of Ownable Contract

Summary

The `Pot` contract incorrectly initializes the `Ownable` parent contract in the inheritance declaration instead of the constructor, which is a wrong statement.

Vulnerability Details

In the `Pot` contract, the `Ownable` parent contract is incorrectly initialized in the inheritance declaration:

```
contract Pot is Ownable(msg.sender) {}
```

This syntax is not supported in Solidity and does not correctly initialize the `Ownable` contract. Although in this specific case, since the `Pot` contract is always deployed by the `ContestManager` contract, the `msg.sender` at deployment will indeed be the `ContestManager`, meaning that the `ContestManager` becomes the owner of the `Pot` contract. Thus, the core functionality remains intact.

Impact

- **Confusion and Misinterpretation**: The incorrect syntax could lead to confusion and incorrect assumptions about the contract's ownership and security model.
- **Potential Introduction of Errors**: Future changes or refactoring might introduce errors if developers assume this pattern is valid and reuse it incorrectly.

Tools Used

Manual Review

Recommendations

The `Ownable` contract should be correctly initialized in the constructor of the `Pot` contract. Even though the `msg.sender` in this context is correctly set to the `ContestManager`, the proper syntax ensures clarity and maintains code quality:

```
- contract Pot is Ownable(msg.sender) {
+ contract Pot is Ownable {
-     constructor(address[] memory players, uint256[] memory rewards,
   IERC20 token, uint256 totalRewards) {
+     constructor(address[] memory players, uint256[] memory rewards,
   IERC20 token, uint256 totalRewards) Ownable() {
          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];
          }
      }
  }
```