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

High Risk Findings

[H-1] Reentrancy in `claimAllRewards`

Description: The function `claimAllRewards` suffers from reentrancy, meaning a malicious smart contract can reenter the functions and withdraw all funds.

Impact: Loss/Withdrawal of all funds from the protocol

Proof of Concept:

1. A user deploys a malicious smart contract that will buy a box and open it until it gets one with a prize
2. User call malicious smart contract to call `claimAllRewards()`
3. The `fallback()` function of the malicious smart contract call `claimAllRewards()` until there is no more fund

Recommended Mitigation:

1. Follow CEI :

```
    function claimAllRewards() public {
        uint256 totalValue = 0;
        for (uint256 i = 0; i < rewardsOwned[msg.sender].length; i++) {
            totalValue += rewardsOwned[msg.sender][i].value;
        }
        require(totalValue > 0, "No rewards to claim");
+       delete rewardsOwned[msg.sender];

        (bool success,) = payable(msg.sender).call{value: totalValue}("");
        require(success, "Transfer failed");

-       delete rewardsOwned[msg.sender];
    }
```

[H-2] Reentrancy in `claimSingleReward()`

Description: The `MysteryBox::claimSingleReward()` function is vulnerable to a reentrancy attack because it doesn't follow the Checks-Effects-Interactions (CEI) pattern. It sends funds to the user **before** deleting the corresponding rewards from the internal state. This vulnerability allows malicious actors who receive at least one Bronze Coin (or higher rewards) to potentially drain the entire contract balance.

Impact: An attacker can exploit this vulnerability to steal all funds collected by the protocol.

Proof of Concept:

1. A user deploys a malicious smart contract that will buy a box and open it until it gets one with a prize
2. User call malicious smart contracts to call `claimSingleReward(indexWinningPrize)`with the index of a winning prize
3. The `fallback()` function of the malicious smart contract call `claimSingleReward(indexWinningPrize)` until there is no more fun

Recommended Mitigation: Follow CEI pattern:

```
    function claimSingleReward(uint256 _index) public {
        require(_index <= rewardsOwned[msg.sender].length, "Invalid index");
        uint256 value = rewardsOwned[msg.sender][_index].value;
        require(value > 0, "No reward to claim");

+.      delete rewardsOwned[msg.sender][_index];

        //@audit possible to reenter here as well
        (bool success,) = payable(msg.sender).call{value: value}("");
        require(success, "Transfer failed");

-       delete rewardsOwned[msg.sender][_index];
    }
```

[H-3] Anyone can change owner

Description: The Contracts owner can change at any time as there are no restrictions on the `changeOwner` function.

Impact: Anyone can change the owner of the contract to an arbitrary address.

Proof of Concept:

1. Add this test to the test suite:

```
function test_changeOwnerIsMissingAccessControl() public {
  // Check if `owner` is the owner of the MysteryBox contract
  assertEq(mysteryBox.owner(), owner);

  // Impersonates a non-owner address `user1`
  vm.startPrank(user1);

  // Give the ownership of the MysteryBox contract to `user2`
  mysteryBox.changeOwner(user2);

  // Stops impersonating `user1`
  vm.stopPrank();

  // Check if `user2` is the new owner of the MysteryBox contract
```

```
    assertEq(mysteryBox.owner(), user2);
  }
```

2. And execute it:

```
forge test --match-test test_changeOwnerIsMissingAccessControl
```

Recommended Mitigation:

1. Modify the `MysteryBox::changeOwner` function, checking if the caller is the `MysteryBox` contract owner:

```
modified   src/MysteryBox.sol
@@ -109,6 +109,7 @@ contract MysteryBox {
     }

     function changeOwner(address _newOwner) public {
+        require(msg.sender == owner, "Only the owner can change the contract
ownership");
         owner = _newOwner;
     }
 }
```

Medium Risk Findings

[M-1] Protocol should have a higher initial balance to prevent prize withdrawing problems

Description: The `MysteryBox.sol#openBox()` function allows users to open their mystery boxes and receive rewards based on the following probabilities:

- 75% chance of getting "Coal"
- 20% chance of getting a "Bronze Coin"
- 4% chance of getting a "Silver Coin"
- 1% chance of getting a "Gold Coin"

When a user opens a box, the number of boxes they own is reduced by 1. However, if a user wins a high-value reward (such as a Gold Coin) and tries to claim it through `MysteryBox.sol#claimAllRewards()`, the contract may not have enough balance to cover the payout. Given that each box costs only 0.1 ETH, it would take thousands of box purchases to accumulate enough ETH to pay out a single Gold Coin reward (1 ETH). As a result, the contract will revert the transaction, leaving the user unable to claim their prize.

```
 function openBox() public {
     //  if contract does not have enough ether to send, it will revert
```

```
        require(boxesOwned[msg.sender] > 0, "No boxes to open");

        // Generate a random number between 0 and 99
        uint256 randomValue = uint256(keccak256(abi.encodePacked(block.timestamp,
msg.sender))) % 100;

        // Determine the reward based on probability
        if (randomValue < 75) {
            // 75% chance to get Coal (0-74)
            rewardsOwned[msg.sender].push(Reward("Coal", 0.1 ether));
        } else if (randomValue < 95) {
            // 20% chance to get Bronze Coin (75-94)
            rewardsOwned[msg.sender].push(Reward("Bronze Coin", 0.1 ether));
        } else if (randomValue < 99) {
            // 4% chance to get Silver Coin (95-98)
            rewardsOwned[msg.sender].push(Reward("Silver Coin", 0.5 ether));
        } else {
            // 1% chance to get Gold Coin (99)
            rewardsOwned[msg.sender].push(Reward("Gold Coin", 1 ether));
        }

        boxesOwned[msg.sender] -= 1;
    }
```

Impact: If a user wins a large number of Bronze Coins or a high-value Gold Coin, they will not be able to claim their rewards due to insufficient contract funds. Every attempt to claim the rewards will revert until the contract has enough balance, leading to frustration and loss of trust in the platform.

Proof of Concept:

1. Several users buy boxes, contributing a total of 0.50 ETH to the contract.
2. A new user wins a Gold Coin (worth 1 ETH).
3. The user attempts to withdraw their reward using `claimAllRewards()`.
4. The transaction reverts because the contract only holds 0.50 ETH and cannot cover the 1 ETH reward.

Recommended Mitigation: To mitigate this issue:

1. Adjust the price of the mystery boxes to ensure the contract can cover potential high-value rewards.
2. Restrict the issuance of high-value rewards unless the contract has sufficient balances.
3. Set aside a portion of every box purchase to build a reserve that can cover large rewards.

[M-2] A user can poison the `rewardsOwned` of another user via `transferReward` of an empty reward index

Description: The `transferReward()` function in the `MysteryBox` contract contains a bug that leaves gaps in the `rewardsOwned` array after transferring a reward. This is due to the use of the `delete` keyword, which only sets the array element to its default value instead of removing it. This behavior results in an array with default values at certain indexes, potentially causing issues during iteration and other operations on the array.

Impact:

1. Leaving default values in the array can cause issues with iteration, indexing, and general array handling, potentially leading to incorrect behavior in other parts of the contract.
2. The presence of "empty" elements in the array may result in wasted gas costs when iterating over the array, or bugs if the code assumes all array elements are valid rewards.

Proof of Concept:

1. The `transferReward()` function currently performs the following steps:

```
rewardsOwned[_to].push(rewardsOwned[msg.sender][_index]);
delete rewardsOwned[msg.sender][_index]; // Leaves a gap
```

2. Key Issue:
    - The `delete` operation sets the reward at `_index` to its default value, but it does not remove the element from the array. This creates a gap in the array where the deleted element still exists, but its values are set to defaults (`""` for strings, `0` for numbers).
    - Arrays in Solidity remain the same size after the `delete` operation, which can lead to unexpected behavior when interacting with the array or iterating over it.

3. Example of an Exploit

    Gaps in the Array Leading to Misuse:

    Initial Setup: Alice has three rewards: `Reward1`, `Reward2`, and `Reward3`. She transfers `Reward2` (at index `1`) to Bob. **Execution**: The contract uses the `delete` keyword to "remove" `Reward2` **Outcome**: Alice's `rewardsOwned` array now looks like:

```
[
    Reward1,
    { name: "", value: 0 },  // Default values left by delete
    Reward3
]
```

NB: This array still contains three elements, but one of them is a default value, creating a gap. This may cause issues when other functions operate on the array, as they may assume that all elements are valid rewards.

Recommended Mitigation:

1. Replace the element at `_index` with the last element in the array, then remove the last element using `.pop()`.

```
function transferReward(address _to, uint256 _index) public {
    require(_index < rewardsOwned[msg.sender].length, "Invalid index");
    rewardsOwned[_to].push(rewardsOwned[msg.sender][_index]);
    // Replace the reward at _index with the last element and pop the last one
    rewardsOwned[msg.sender][_index] = rewardsOwned[msg.sender]
[rewardsOwned[msg.sender].length - 1];
    rewardsOwned[msg.sender].pop();
}
```

[M-3] Weak Randomness

Description: In MysteryBox.sol:openBox function on line 48 the there is a randomness that is weak and can be potentially guessed.

Impact: Impact is that users may be able to get Gold Coin on every play if they are running calculations.

Proof of Concept: When you are taking the keccack256 of a block.timestamp and msg.sender you are able to maniuplate the possible results by either calculating a specific time to win the prize you want or by changing the msg.sender until you get the results that you want.

Recommended Mitigation: Take the randomness off chain using something like the Chainlink Verifiable Random Function.

[M-4] addReward won't have any effect on openBox

Description: The `addReward()` and `openBox()` functions in the `MysteryBox` contract introduce an issue where newly added rewards are not being properly distributed to users. After calling `addReward()`, the new rewards cannot be assigned to any user when opening a box, which breaks the reward distribution logic.

Impact:

1. Newly added rewards are inaccessible.

2. Inconsistent reward distribution.

Proof of Concept:

1. `addReward()`: This function allows the contract owner to add new rewards to the reward pool. However, the logic in the `openBox()` function does not dynamically account for the added rewards, meaning newly added rewards are not included in the reward assignment process.

2. `openBox()`: This function generates a random value to assign predefined rewards based on fixed conditions (coal, bronze, silver, and gold). It does not dynamically update to accommodate newly added rewards from the `addReward()` function, effectively rendering those rewards unavailable to users.

Recommended Mitigation:

1. Modify the `openBox()` function to dynamically select rewards from the `rewardPool` array. This can be done by introducing a weighted random selection based on the size of the reward pool.

2. By dynamically adjusting the reward distribution logic, the contract will correctly handle any rewards added by the owner and will ensure users can receive them.

Low Risk Findings

[L-1] The rewards in constructor are different from the rewards in openBox

Description: The `openBox` function doesn't match the reward pool defined in the constructor.

Impact: This mismatch can lead to an unfair distribution of rewards and potential economic imbalance in the game.

Proof of Concept:

```
    constructor() payable {
        owner = msg.sender;
        boxPrice = 0.1 ether;
        require(msg.value >= SEEDVALUE, "Incorrect ETH sent");
        // Initialize with some default rewards
        rewardPool.push(Reward("Gold Coin", 0.5 ether));
        rewardPool.push(Reward("Silver Coin", 0.25 ether));
        rewardPool.push(Reward("Bronze Coin", 0.1 ether));
        rewardPool.push(Reward("Coal", 0 ether));
    }

    // inside openBox function
     if (randomValue < 75) {
          // 75% chance to get Coal (0-74)
          rewardsOwned[msg.sender].push(Reward("Coal", 0 ether));
      } else if (randomValue < 95) {
          // 20% chance to get Bronze Coin (75-94)
          rewardsOwned[msg.sender].push(Reward("Bronze Coin", 0.1 ether));
      } else if (randomValue < 99) {
          // 4% chance to get Silver Coin (95-98)
          rewardsOwned[msg.sender].push(Reward("Silver Coin", 0.5 ether));
      } else {
          // 1% chance to get Gold Coin (99)
          rewardsOwned[msg.sender].push(Reward("Gold Coin", 1 ether));
      }
```

Recommended Mitigation:

1. Consider using the constant/variables to replace the literal/magic number. It can make code much easier to read and make sure they stay the same value.

[L-2] Gas Limit Exhaustion in `claimAllRewards` Function

Description: The `claimAllRewards()` function may run out of gas if a user has a large number of rewards, preventing them from claiming all their rewards in a single transaction.

Impact: Users with many rewards may not be able to claim all of their rewards in one transaction due to gas limits, leading to user frustration and unclaimed rewards.

Proof of Concept:

1. The `claimAllRewards()` function loops through all rewards a user has, which may exceed the block gas limit for users with many rewards:

```
for (uint256 i = 0; i < rewardsOwned[msg.sender].length; i++) {
    totalValue += rewardsOwned[msg.sender][i].value;
}
```

Recommended Mitigation: Implement a batched reward claiming system where users can claim rewards in smaller batches across multiple transactions to avoid gas limit issues.