Protocol Audit Report

Jordan J. Solomon

October 09th, 2024

CodeHawks First Flight

Version 1.0

JJS

October 9, 2024

Protocol Audit Report

Jordan J. Solomon

October 09th, 2024

Prepared by: JJS

Table of Contents

Protocol Summary

MysteryBox is a thrilling protocol where users can purchase mystery boxes containing random rewards! Open your box to reveal amazing prizes, or trade them with others. Will you get lucky and find the rare treasures?

Disclaimer

JJS will make 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

Scope

src/ – MysteryBox.sol

Roles

- Owner/Admin (Trusted) - Can set the price of boxes, add new rewards, and withdraw funds.
- User/Player - Can purchase mystery boxes, open them to receive rewards, and trade rewards with others.

Executive Summary

Issues found

Severity	Issues Found
High	2
Medium	2
Low	1
Info	4

Findings

High

[H-1] Missing access control on `MysteryBox::changeOwner`

Summary

In the MysteryBox.sol contract many functions rely on a require that checks if the msg.sender is in fact the owner. These functions include: - setBoxPrice - Setting the box price
- addReward - Adding a whole new reward - withdrawFunds - Withdrawing the funds from the contract All of these but not the changeOwner function, which changes the address of the owner!

Vulnerability Details

Missing access control checks on the changeOwner function which changes the address of the owner:

```
function changeOwner(address _newOwner) public {
    //* Missing code here! *//
    owner = _newOwner;
}
```

Impact

This vulnerability could potentially lead to loss of all funds in the contract. Consider the following scenario and test that can be added to testMysteryBox.t.sol:

1. The owner deployed with 0.1 ether (in the setup)
2. Two users buy boxes
3. An attacker sees the vulnerability
4. The attacker changes himself to be the owner
5. The attacker (now the owner of the contract) withdraws the funds

PoC

```
function test_can_steal_funds_as_owner() public {
        uint256 amount = 0.1 ether;
        vm.deal(user1, amount);
        vm.deal(user2, amount);
        vm.prank(user1);
        mysteryBox.buyBox{value: amount}();
        vm.prank(user2);
        mysteryBox.buyBox{value: amount}();

        address attacker = makeAddr("attacker");
```

```
        vm.startPrank(attacker);
        mysteryBox.changeOwner(attacker);
        mysteryBox.withdrawFunds();
        vm.stopPrank();

        assertEq(attacker.balance, amount * 3); // 0.1 ether * 3 from user1, use
    }
```

Tools Used

Manual Review, Unit Test

Recommendations

Add an onlyOwner modifier (then can be added to the necessary functions mentioned earlier):

```
+ modifier onlyOwner() {
+    if (msg.sender != owner){
+        revert MysteryBox__NotOwner;
+    }
+    _;
+ }
```

Or can just add a check in changeOwner for owner:

```
function changeOwner(address _newOwner) public {
+    require (msg.sender == owner, "Not Owner!");
     owner = _newOwner;
}
```

[H-2] Reentrancy In `MysteryBox::claimAllRewards` can potentially end in loss of funds

Summary

The claimAllRewards function lets a user claim all of their rewards in one transaction. However, the fundamentals of writing function is Solidity is forgot in this function - The Checks-Effects-Interactions. Forgetting to do so can lead to reentrancy attack in the protocol.

Vulnerability Details

Code Snippet

```
    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");

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

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

Impact

Reentrancy attacks are dangerous to protocols. Because the state is not yet updated. They can end up with drained contracts of funds! Consider the following scenario:

1. User has bought a box.
2. Then opened it.
3. The user continued to claim all his rewards but has a malicious fallback-/receive function.
4. Because the state is not yet update (the mapping element is not yet deleted) there is an attack opening.
5. His fallback/receive function calls back into claimAllRewards.
6. The function sends them another payment.
7. The attacker keeps on until they drain the funds out of the contract.

Tools Used

Manual Review

Recommendations

Follow CEI (Checks-Effects-Interactions) to prevent this attack:

```
    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");

+       (bool success,) = payable(msg.sender).call{value: totalValue}("");
```

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

Medium

[M-1] Weak Randomness in `MysteryBox::openBox`

Summary

The openBox function is in charge of determining the reward that the user receives upon opening the box they bought. It does that but calculating a random number and depending on that number is the reward that the user gets. However, determining the random number on chain is not possible like such. Hashing the block.timestamp and the address of (msg.sender) will create a predictable number on-chain.

Vulnerability Details

Code Snippet

```
    function openBox() public {
        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
<@


        // Determine the reward based on probability
        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));
        }

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

Impact

- Malicious users can manipulate those values, or know what they will be, helping them choose a user the win.
- This also lets users front-run and requesting a refund if they are not the winner.

Tools Used

Manual Review

Recommendations

Consider using a cryptographically provable random number generator such as Chainlink VRF.

[M-2] Deleting elements in the mapping will not change the array length, leading to loss of funds in `MysteryBox::transferReward`

Summary

The transferReward function will transfer a msg.sender's rewards to another user. The last line of the function deletes the rewards in a certain index, because it has been transferred to the other user. However, the function relies on the length of that array to make sure that the index is in it. Deleting the element in the mapping will not shorten the length. Thus a malicious user could pass in the same index an still transfer rewards that are not theirs anymore.

Vulnerability Details

Code Snippet

```
function transferReward(address _to, uint256 _index) public {
@>      require(_index < rewardsOwned[msg.sender].length, "Invalid index");
<@
        rewardsOwned[_to].push(rewardsOwned[msg.sender][_index]);
@>      delete rewardsOwned[msg.sender][_index];    <@
    }
```

Impact

This will cause errors when the code will go through the require line. Consider this scenario and test that can be implemented in the test file:

1. Alice transfers her rewards to Bob.
2. The function will transfer the rewards.
3. Then it will delete the elements. Not changing the length of the array.

4. Alice passes the same index to the function. It will pass the require.
5. Alice transfers more rewards, the same one she already transferred.
6. Bob can withdraw the funds that he shouldn't have

PoC

```
function test_can_steal_funds_with_same_index() public {
    // Set up addresses
    address alice = makeAddr("alice");
    address bob = makeAddr("bob");

    // Give funds to Alice and buying boxes
    vm.deal(alice, 0.2 ether);
    vm.startPrank(alice);
    mysteryBox.buyBox{value: 0.1 ether}();
    mysteryBox.buyBox{value: 0.1 ether}();
    // Opening boxes for reward
    mysteryBox.openBox();
    mysteryBox.openBox();
    // Transfer rewards to Bob
    mysteryBox.transferReward(bob, 0);
    mysteryBox.transferReward(bob, 0);
    vm.stopPrank();
}
```

Tools Used

Manual review, unit test

Recommendations

Consider using a nested mapping instead of array, this way there is no reason to check for length:

```
- mapping(address => Reward[]) public rewardsOwned;
- Reward[] public rewardPool;
+ mapping(address => mapping(Reward => uint256)) public s_rewardsOwned;
```

Low

[L-1] Use of Magic Numbers Leads to Wrong Reward Distribution in `MysteryBox::openBox`

Summary

the openBox function is in charge of determining the reward that the user receives upon opening the box they bought. It does that but calculating a

random number and depending on that number is the reward that the user gets. However, because of use of magic number (hard coding numbers in the function rather than saving them in a variable), the rewards are wrong for the silver and gold rewards:

Vulnerability Details

Code Snippet

```
    function openBox() public {
        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

        // Determine the reward based on probability
        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));
<@
        }

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

Impact

This means that for every user that receives the silver or gold coin rewards they get double what they were meant to get! In the constructor it is clearly not those numbers:

Code Snippet

```
    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));
    }
```

Tools Used

Manual Review

Recommendations

Use constant variables instead of magic number:

Code Diff

```
+ uint256 constant GOLD_PRICE = 0.5 ether;
+ uint256 constant SILVER_PRICE = 0.25 ether;
+ uint256 constant BRONZE_PRICE = 0.1 ether;

function openBox() public {

    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, ms

    // Determine the reward based on probability
    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));
+       rewardsOwned[msg.sender].push(Reward("Bronze Coin", BRONZE_PRICE));
    } else if (randomValue < 99) {
        // 4% chance to get Silver Coin (95-98)
-       rewardsOwned[msg.sender].push(Reward("Silver Coin", 0.5 ether));
+       rewardsOwned[msg.sender].push(Reward("Silver Coin", SILVER_PRICE));
    } else {
        // 1% chance to get Gold Coin (99)
-       rewardsOwned[msg.sender].push(Reward("Gold Coin", 1 ether));
+       rewardsOwned[msg.sender].push(Reward("Gold Coin", GOLD_PRICE));
    }
```

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

Info

[I-1] Floating versions of solidity

Summary

The smart contract uses a floating (inexact) version of the Solidity compiler in its pragma statement. For example:

```
pragma solidity ^0.8.0;
```

A floating version (^0.8.0) allows the contract to be compiled with any version of the Solidity compiler that is 0.8.x, meaning it will accept any minor or patch update within that range. While this provides flexibility and allows the contract to be compiled with newer compiler versions (which may include important bug fixes or performance improvements), it also introduces the risk of unanticipated behavior or vulnerabilities due to changes in Solidity's behavior in future minor versions.

Impact

Floating version pragmas can lead to potential issues if a future Solidity compiler version introduces changes that affect the contract's functionality. New versions of Solidity may change internal optimizations, error handling, or even introduce new bugs that did not exist in previous versions. This could make the contract behave differently than expected, especially if the new compiler introduces breaking changes.

Tools Used

Manual Review

Recommendations

It is recommended to use a fixed Solidity version in the pragma statement to ensure the contract always compiles with the exact version of the Solidity compiler it was tested with.

[I-2] Missing Best Practice Variable Namings

Summary

When writing smart contracts the naming of the variable is important. The name should show what type of variable it is:

- s_name for state variables
- i_name for immutable variables
- NAME for constant variables

Vulnerability Details

Here is the list of variables that lack the naming conventions:

Code

```
address public owner;
uint256 public boxPrice;
mapping(address => uint256) public boxesOwned;
mapping(address => Reward[]) public rewardsOwned;
Reward[] public rewardPool;
```

Impact

This leads to a worse understanding when reading the code, could lead to miss use for development

Tools Used

Manual Review

Recommendations

Change the names to the proper convention:

Diff

```
- address public owner;
+ address public s_owner;
- uint256 public boxPrice;
+ uint256 public s_boxPrice;
- mapping(address => uint256) public boxesOwned;
+ mapping(address => uint256) public s_boxesOwned;
- mapping(address => Reward[]) public rewardsOwned;
+ mapping(address => Reward[]) public s_rewardsOwned;
- Reward[] public rewardPool;
```

[I-3] Missing Events

Summary

For important scenarios, emitting an event is important for the user and the system.

Vulnerability Details

Missing functions that need events:

- setBoxPrice
- addReward
- buyBox
- openBox
- withdrawFunds
- transferReward
- claimAllRewards
- claimSingleReward
- changeOwner

Impact

Users and the owner can be confused by occurrences in the transactions.

Tools Used

Manual Review

Recommendations

Add events in these functions.

[I-4] No Use of Custom Errors