Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2023.11.15, the SlowMist security team received the Airian team's security audit application for Airian, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project team should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

Serial Number	Audit Class	Audit Subclass
1	Overflow Audit	-
2	Reentrancy Attack Audit	-
3	Replay Attack Audit	-
4	Flashloan Attack Audit	-
5	Race Conditions Audit	Reordering Attack Audit
6	Permission Vulnerability Audit	Access Control Audit
		Excessive Authority Audit
7	Security Design Audit	External Module Safe Use Audit
		Compiler Version Security Audit
		Hard-coded Address Security Audit
		Fallback Function Safe Use Audit
		Show Coding Security Audit
		Function Return Value Security Audit
		External Call Function Security Audit

Serial Number	Audit Class	Audit Subclass
7	Security Design Audit	Block data Dependence Security Audit
		tx.origin Authentication Security Audit
8	Denial of Service Audit	-
9	Gas Optimization Audit	-
10	Design Logic Audit	-
11	Variable Coverage Vulnerability Audit	-
12	"False Top-up" Vulnerability Audit	-
13	Scoping and Declarations Audit	-
14	Malicious Event Log Audit	-
15	Arithmetic Accuracy Deviation Audit	-
16	Uninitialized Storage Pointer Audit	-

3 Project Overview

3.1 Project Introduction

Audit Version:

https://github.com/airian-io/airian-contracts

commit: 1b81b8ea196bfdf11b224fa16b4c7b193dc0b531

(Does not include the code content of the comment part)

Fixed Version:

https://github.com/airian-io/airian-contracts

commit: 9da5ec65ecb312de689b616b2e5548e567854422

(Does not include the code content of the comment part)

3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	Overwrite depositIndex issue	Variable Coverage Vulnerability	High	Fixed
N2	Unseparated depositList record	Design Logic Audit	High	Fixed
N3	Risk of excessive authority	Authority Control Vulnerability Audit	Medium	Fixed
N4	buyItemCredit function lacks the buy logic	Design Logic Audit	Medium	Fixed
N5	Design logic issue	Design Logic Audit	Medium	Fixed
N6	Receive Ether can lock users' native tokens	Others	Low	Fixed
N7	Redundant require check	Others	Low	Fixed
N8	Missing check return value	Others	Low	Fixed
N9	Using the transfer function to transfer ETH may cause assets to be locked	Others	Low	Fixed
N10	Missing the checks	Others	Suggestion	Fixed
N11	Gas Optimization	Gas Optimization Audit	Suggestion	Fixed
N12	Redundant code	Others	Suggestion	Fixed
N13	Failure to follow the Checks-Effects-Interactions principle	Replay Vulnerability	Suggestion	Fixed
N14	Random number check	Others	Suggestion	Fixed

4 Code Overview

4.1 Contracts Description

This is the Airian project. The project contains the ERC20 token and ERC721 token parts. The Air contract is the ERC20 token contract, the owner can call the mint functions to mint AIR tokens and there is an upper limit for the token amount. In the AirDrop, Collection, EvenAllocation, MysteryBox, and PfpCollection contracts, users can choose to use the native tokens or the quote tokens to purchase the NFT and claim. And users can stake the native token or the quote token and the whiteList NFT in the Subscription contract.

The main network address of the contract is as follows:

The code was not deployed to the mainnet.

4.2 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

Air			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
mint	External	Can Modify State	onlyOwner whenNotPaused
transfer	External	Can Modify State	whenNotFrozen whenNotPaused checkLock
transferFrom	External	Can Modify State	whenNotFrozen whenNotPaused checkLock
approve	External	Can Modify State	-
name	External	-	-
symbol	External	-	-
decimals	External	-	-

WhiteListNFT			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC721
pause	Public	Can Modify State	onlyRole
unpause	Public	Can Modify State	onlyRole
addMinter	Public	Can Modify State	onlyOwner
addPauser	Public	Can Modify State	onlyOwner
safeMint	Public	Can Modify State	onlyRole
safeBatchMint	Public	Can Modify State	onlyRole
safeBatchMintLight	Public	Can Modify State	onlyRole
_beforeTokenTransfer	Internal	Can Modify State	whenNotPaused
_burn	Internal	Can Modify State	-
tokenURI	Public	-	-
supportsInterface	Public	-	-
setStaking	Public	Can Modify State	onlyOwner
addWhitelist	Public	Can Modify State	onlyOwner
safeBatchMintToWhitelist	Public	Can Modify State	onlyRole
getWhitelist	Public	-	-

AirDrop			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC721Token
setConfig	Public	Can Modify State	onlyOwner
setSoulbound	Public	Can Modify State	onlyOwner

AirDrop			
setLaunch	Public	Can Modify State	onlyOwner
setLockup	Public	Can Modify State	onlyOwner
setQuote	Public	Can Modify State	onlyOwner
setMandatory	Public	Can Modify State	onlyOwner
safeMint	Public	-	onlyRole
mintByTokenId	Internal	Can Modify State	-
safeBatchMint	Public	-	onlyRole
registerItems	Public	Can Modify State	onlyRole isNotRegistered
claim	Public	Can Modify State	-
buyItemCredit	Public	Can Modify State	-
<Receive Ether>	External	Payable	-
onERC721Received	Public	-	-
getItemRemains	Public	-	-
_beforeTokenTransfer	Internal	Can Modify State	whenNotPaused

ERC721Token			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC721
pause	Public	Can Modify State	onlyRole
unpause	Public	Can Modify State	onlyRole
addMinter	Public	Can Modify State	onlyOwner
addPauser	Public	Can Modify State	onlyOwner
setAwsKms	Public	Can Modify State	onlyOwner

ERC721Token			
safeMint	Public	Can Modify State	onlyRole
safeMintTo	Public	Can Modify State	onlyRole
mintByClaim	Internal	Can Modify State	-
mintByTokenId	Internal	Can Modify State	-
safeBatchMint	Public	Can Modify State	onlyRole
safeBatchMintLight	Public	Can Modify State	onlyRole
_beforeTokenTransfer	Internal	Can Modify State	whenNotPaused
_burn	Internal	Can Modify State	-
tokenURI	Public	-	-
supportsInterface	Public	-	-
setHardCap	Public	Can Modify State	onlyOwner
setMysteryBox	Public	Can Modify State	onlyOwner

ERC20Lockable			
Function Name	Visibility	Mutability	Modifiers
_lock	Internal	Can Modify State	-
_unlock	Internal	Can Modify State	-
unlock	External	Can Modify State	-
unlockAll	External	Can Modify State	-
releaseLock	External	Can Modify State	onlyOwner
transferWithLockUp	External	Can Modify State	onlyOwner
lockInfo	External	-	-
totalLocked	External	-	-

ERC20			
Function Name	Visibility	Mutability	Modifiers
_transfer	Internal	Can Modify State	-
_approve	Internal	Can Modify State	-
_mint	Internal	Can Modify State	-
_burn	Internal	Can Modify State	-
totalSupply	External	-	-
balanceOf	External	-	-
allowance	External	-	-
name	External	-	-
symbol	External	-	-
decimals	External	-	-
transfer	External	Can Modify State	-
transferFrom	External	Can Modify State	-
approve	External	Can Modify State	-

Ownable			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
owner	External	-	-
transferOwnership	Public	Can Modify State	onlyOwner
renounceOwnership	External	Can Modify State	onlyOwner
_transferOwnership	Internal	Can Modify State	-

ERC20Burnable			
Function Name	Visibility	Mutability	Modifiers
burn	External	Can Modify State	whenNotPaused
burnFrom	External	Can Modify State	whenNotPaused

Pausable			
Function Name	Visibility	Mutability	Modifiers
pause	External	Can Modify State	onlyOwner whenNotPaused
unPause	External	Can Modify State	onlyOwner whenPaused
paused	External	-	-

ERC20Mintable			
Function Name	Visibility	Mutability	Modifiers
mint	External	Can Modify State	-
finishMint	External	Can Modify State	onlyOwner
isFinished	External	-	-

Freezable			
Function Name	Visibility	Mutability	Modifiers
freeze	External	Can Modify State	onlyOwner
unFreeze	External	Can Modify State	onlyOwner
isFrozen	External	-	-

Subscription			
Function Name	Visibility	Mutability	Modifiers

Subscription			
<Constructor>	Public	Can Modify State	-
stakingEth	Public	Payable	initialized isNotAllocated nonReentrant
stakingQuote	Public	Can Modify State	initialized isNotAllocated nonReentrant
_stakingNFT	Private	Can Modify State	-
_refundNFT	Private	Can Modify State	-
unStaking	Public	Can Modify State	isNotAllocated nonReentrant
<Receive Ether>	External	Payable	-
estimateRandomizeFee	External	-	-
requestRandomNumber	External	Payable	-
_fetchRandomNumber	Private	Can Modify State	-
_getIndexByAddress	Private	-	-
claim	External	Can Modify State	nonReentrant
_refund	Private	Can Modify State	-
setWhiteList	Public	Can Modify State	onlyOwner
setConfig	Private	Can Modify State	onlyOwner
onERC721Received	Public	-	-
getAllocated	Public	-	-
getLeastFund	Public	-	-
getParticipants	Public	-	-
getMyFund	Public	-	-

Subscription			
getBooking	Public	-	-
getMyWin	Public	-	-
getClaimed	Public	-	-

MysteryBox			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC721Token
setMboxKey	Public	Can Modify State	onlyOwner
setConfig	Public	Can Modify State	onlyOwner
setLaunch	Public	Can Modify State	onlyOwner
setLockup	Public	Can Modify State	onlyOwner
setQuote	Public	Can Modify State	onlyOwner
safeMint	Public	-	onlyRole
mintByClaim	Internal	Can Modify State	-
safeBatchMint	Public	-	onlyRole
registerItems	Public	Can Modify State	onlyRole isNotRegistered
claim	Public	Can Modify State	whenNotPaused nonReentrant
buyItemCredit	Public	Can Modify State	whenNotPaused nonReentrant
<Receive Ether>	External	Payable	-
estimateRandomizeFee	External	-	-
requestRandomNumber	External	Payable	-
_fetchRandomNumber	Private	Can Modify State	-
_deleteItemAfterMint	Private	Can Modify State	-

MysteryBox			
buyKeyEth	Public	Payable	-
buyKeyQuote	Public	Can Modify State	-
buyKeyCredit	Public	Can Modify State	onlyRole
onERC721Received	Public	-	-
settlement	Private	Can Modify State	-
getTokenURI	Private	Can Modify State	-
getItemRemains	Public	-	-
setMandatory	Public	Can Modify State	onlyOwner

EvenAllocation			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
buyTicketEth	Public	Payable	initialized isNotAllocated nonReentrant
buyTicketQuote	Public	Can Modify State	initialized isNotAllocated nonReentrant
_stakingNFT	Private	Can Modify State	-
_refundNFT	Private	Can Modify State	-
<Receive Ether>	External	Payable	-
estimateRandomizeFee	External	-	-
requestRandomNumber	External	Payable	-
_fetchRandomNumber	Private	Can Modify State	-
_getIndexByAddress	Private	-	-

EvenAllocation			
claim	External	Can Modify State	nonReentrant
_refund	Private	Can Modify State	-
setWhiteList	Public	Can Modify State	onlyOwner
setConfig	Private	Can Modify State	onlyOwner
onERC721Received	Public	-	-
getAllocated	Public	-	-
getMyTickets	Public	-	-
getMyWin	Public	-	-
getTicketCount	Public	-	-
getParticipants	Public	-	-
getBooking	Public	-	-
getClaimed	Public	-	-

Collection			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC721Token
setConfig	Public	Can Modify State	onlyOwner
setLaunch	Public	Can Modify State	onlyOwner
setLockup	Public	Can Modify State	onlyOwner
setQuote	Public	Can Modify State	onlyOwner
safeMint	Public	-	onlyRole
mintByTokenId	Internal	Can Modify State	-

Collection			
safeBatchMint	Public	-	onlyRole
registerItems	Public	Can Modify State	onlyRole isNotRegistered
buyItemEth	Public	Payable	-
buyItemQuote	Public	Can Modify State	-
buyItemCredit	Public	Can Modify State	onlyRole
settlement	Private	Can Modify State	-
<Receive Ether>	External	Payable	-
onERC721Received	Public	-	-
getItemRemains	Public	-	-
setMandatory	Public	Can Modify State	onlyOwner

4.3 Vulnerability Summary

[N1] [High] Overwrite depositIndex issue

Category: Variable Coverage Vulnerability

Content

In the EvenAllocation contract, users can call the buyTicketEth and buyTicketQuote functions to purchase the ticket, and the `depositIndex` will record by the current `nDeposit`. Once called the buyTicketEth and buyTicketQuote function, the current `nDeposit` will add 1. The `nTickets` and `deposit` parameters will be recorded in the booking array. If the user calls the buyTicketEth and buyTicketQuote functions multiple times to buy the tickets, his `depositIndex` will be recorded as the latest `nDeposit`. When the user calls the claim function to claim the NFT and refund, the `depositIndex` will use the latest booking array from this user, and the purchase recorded in the booking array before will lock in the array. And the deposited funds can not claim the NFT or refund.

Code location:

EvenAllocation.sol#

```
function buyTicketEth(uint256 _nTickets)
    public
    payable
    initialized
    isNotAllocated
    nonReentrant
{
    ......
    depositList[msg.sender] = depositList[msg.sender].add(msg.value);

    depositIndex[msg.sender] = nDeposit.current();
    booking.push(
        Book({
            user: msg.sender,
            nTickets: _nTickets,
            deposit: msg.value,
            allocated: 0,
            status: false,
            claimed: false
        })
    );
    for (uint256 i = 0; i < _nTickets; i++) {
        ticketData.push(nDeposit.current());
    }
    ticketSold = ticketSold.add(_nTickets);
    remainTickets = ticketSold;
    nDeposit.increment();
    ......
}

function claim() external nonReentrant {
    ...
    uint256 ix = depositIndex[msg.sender];
    ...
}
```

Solution

It is recommended to clarify the business design to complete the correspondence between the booking array and depositIndex data.

Status

Fixed

[N2] [High] Unseparated depositList record

Category: Design Logic Audit

Content

In the EvenAllocation and Subscription contracts, users can call the buyTicketEth/buyTicketQuote and stakingEth/stakingQuote functions to purchase tickets or staking tokens. All the deposits will be recorded in the `depositList` parameter, no matter the native token or the quote token. So there is a situation in which when the users deposit the native token or the quote token first, the owner role changes the deposit method. All the native tokens and the quote token will be recorded in the `depositList`. And if the price of the native token and quote token are different, the refund will be wrong.

Code location:

Subscription.sol#172-291

EvenAllocation.sol#162-297

```
    function stakingEth()
        public
        payable
        initialized
        isNotAllocated
        nonReentrant
    {
        ......
        depositList[msg.sender] = depositList[msg.sender].add(msg.value);
        ......
    }

    // Before this Approve on quote token is needed
    function stakingQuote(uint256 _payment)
        public
        initialized
        isNotAllocated
        nonReentrant
    {
        ......
        depositList[msg.sender] = depositList[msg.sender].add(_payment);
        ......
```

```
    }

    function buyTicketEth(uint256 _nTickets)
        public
        payable
        initialized
        isNotAllocated
        nonReentrant
    {
        ......
        depositList[msg.sender] = depositList[msg.sender].add(msg.value);
        ......
    }

    function buyTicketQuote(uint256 _payment, uint256 _nTickets)
        public
        initialized
        isNotAllocated
        nonReentrant
    {
        ......
        depositList[msg.sender] = depositList[msg.sender].add(_payment);
        ......
    }
```

Solution

It is recommended to separate the deposit recording methods of quote token and native token.

Status

Fixed

[N3] [Medium] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

1.In the ERC20Lockable contract, the owner can release the locked tokens of any user through the releaseLock function. If some tokens are unlocked unexpectedly, it will lead to unknown market risks.

Code location:

erc20/ERC20Lockable.sol#66-77

```
    function releaseLock(address from)
    external
```

```
    onlyOwner
    returns (bool success)
    {
        for(uint256 i = 0; i < _locks[from].length; i++){
            if(_unlock(from, i)){
                i--;
            }
        }
        success = true;
    }
```

2.In the AirDrop, Collection, EvenAllocation, MysteryBox, and Subscription contracts, the owner role can modify some sensitive parameters that can affect users' assets or in the case of purchase and payment.

Code location:

AirDrop.sol#87-132

Collection.sol#81-108

EvenAllocation.sol#625-684

MysteryBox.sol#131-201

Subscription.sol710-762

3.In the ERC721Token contract, the owner role can modify the hardCap, globalURI, and mysteryBox parameters from the setHardCap and setMysteryBox functions.

Code location:

ERC721Token.sol#179-189

```
    function setHardCap(uint256 _hardCap, string memory _uri) public onlyOwner {
        hardCap = _hardCap;
        globalURI = _uri;
        emit SetHardCap(hardCap);
    }

    function setMysteryBox(address _mbox) public onlyOwner {
        mysteryBox = _mbox;
        _grantRole(MINTER_ROLE, _mbox);
        emit SetMysteryBox(mysteryBox);
    }
```

4.In the WhiteListNFT contract, the owner role can add a new minter and the minter role can mint NFT arbitrarily without an upper limit. The owner role can also modify the staking address and whitelist.

Code location:

WhiteListNFT.sol#69-72, 79-110,167-198

Solution

In the short term, transferring owner ownership to multisig contracts is an effective solution to avoid single-point risk. But in the long run, it is a more reasonable solution to implement a privilege separation strategy and set up multiple privileged roles to manage each privileged function separately. And the authority involving user funds should be managed by the community, and the authority involving emergency contract suspension can be managed by the EOA address. This ensures both a quick response to threats and the safety of user funds.

Status

Fixed

[N4] [Medium] buyItemCredit function lacks the buy logic

Category: Design Logic Audit

Content

In the AirDrop contract, users can call the buyItemCredit function to mint ERC721 tokens and there are not no authority controls or any purchase logic in this function. So anyone can call the function to mint ERC721 tokens.

Code location:

AirDrop.sol#221-263

```
    function buyItemCredit(
        uint256 _itemIx,
        uint256 amount,
        address to
    ) public {
        require(block.timestamp > launch, "Not yet launch time reached");
        require(totalItems > totalSold, "aridrop sold out");
        ......
        for (uint256 i = 0; i < amount; i++) {
            uint256 tokenId = startId.add(_tokenIdCounter[_itemIx].current());
            mintByTokenId(to, tokenId, itemURIs[_itemIx]);
            _tokenIdCounter[_itemIx].increment();
            itemSolds[_itemIx] = itemSolds[_itemIx].add(1);
```

```
            totalSold = totalSold.add(1);
        }
    }
```

Solution

It's recommended to clarify the business logic or authority controls of this function.

Status

Fixed

[N5] [Medium] Design logic issue

Category: Design Logic Audit

Content

In the EvenAllocation and Subscription contract, users can call the buyTicketEth/ buyTicketQuote and stakingEth/stakingQuote functions to purchase the tickets. After the purchase, users can claim from the claim function. But there is no end time check or the claimed status check. If a user purchased the tickets, then he claimed from the claim function, and the claim status of his booking will be set to true without popping the booking array out. Then he calls the purchase function again, the funds will be sent to this contract and he can not call the claim function anymore, which means his funds will lock in the contract.

Code location:

EvenAllocation.sol#167-336

Subscription.sol#177-306

```
    function buyTicketEth(uint256 _nTickets)
        public
        payable
        initialized
        isNotAllocated
        nonReentrant
    {
        require(config.quote == address(0), "Wrong quote token");
        require(
            block.timestamp >= config.launch,
            "Ticket sales is not started"
        );
        require(
            _nTickets > 0,
            "Number of tickets for purchasing should be greater than zero"
```

```
        );
        require(msg.value > 0, "Payment should be greater than zero");
        ...
    }

    function buyTicketQuote(uint256 _payment, uint256 _nTickets)
        public
        initialized
        isNotAllocated
        nonReentrant
    {
        require(config.quote != address(0), "Wrong quote token");
        require(
            block.timestamp >= config.launch,
            "Ticket sales is not started"
        );
        require(
            _nTickets > 0,
            "Number of tickets for purchasing should be greater than zero"
        );
        require(_payment > 0, "Payment should be greater than zero");
        ...
    }
```

Solution

It's recommended to add the end time check or the claim status check of the purchase. If the project team wants the users to participate in the purchase and claim multiple times, and it is also recommended to pop the booking array out after the claim (the pop part for the EvenAllocation contract functions).

Status

Fixed

[N6] [Low] Receive Ether can lock users' native tokens

Category: Others

Content

There is a receive function AirDrop, and Collection contracts so that the contracts can receive native tokens. However, the receive function can lock users' native tokens when users transfer the native token in these contracts by mistake. And the payable modifier can help these functions which need to call with the native tokens.

Code location:

AirDrop.sol#365

Collection.sol#351

```
    receive() external payable {}
```

Solution

It's recommended to remove the redundant receive() function if the contract only needs to receive ether from functions in the contract, and use the payable modifier in these functions instead.

Status

Fixed

[N7] [Low] Redundant require check

Category: Others

Content

1.In the Collection contract, users can call the buyItemEth and the buyItemQuote functions to buy the ERC721 NFT, and these functions check the `require(_payment == _amount.mul(itemPrices[_itemIx]), "value and amount is not match");` and `require(itemAmounts[_itemIx] >= itemSolds[_itemIx].add(_amount), "lack of items left");` checks make the `require(itemAmounts[_itemIx] > itemSolds[_itemIx], "item sold out");` check as a redundant check to consume more gas in the contract.

Code location:

Collection.sol#166,221

```
    function buyItemQuote(
        uint256 _itemIx,
        uint256 _payment,
        uint256 _amount
    ) public {
        require(quote != address(0), "Wrong quote token");
        require(block.timestamp > launch, "Not yet launch time reached");
        require(_payment > 0, "value should be greater than zero");
        require(
            _payment == _amount.mul(itemPrices[_itemIx]),
```

```
            "value and amount is not match"
        );

        uint256 balance = IERC20(quote).balanceOf(msg.sender);
        require(balance > _payment, "lack of quote token balance");
        require(itemAmounts[_itemIx] > itemSolds[_itemIx], "item sold out");
        require(
            itemAmounts[_itemIx] >= itemSolds[_itemIx].add(_amount),
            "lack of items left"
        );
        ......
    }
```

2.In the EvenAllocation contract, users can call the buyTicketEth and the buyTicketQuote functions to buy the ERC721 NFT, and these functions check the `require(_payment > 0, "Payment should be greater than zero");` and `require(config.price.mul(_nTickets) == _payment,"Wrong number of tickets to purchase");` checks make the `require(msg.value >= config.price, "Under least staking amount");` and `require(_payment >= config.price, "Under least staking amount");` check as a redundant check to consume more gas in the contract.

Code location:

EvenAllocation.sol#181,247

```
    function buyTicketEth(uint256 _nTickets)
        public
        payable
        initialized
        isNotAllocated
        nonReentrant
    {
        require(config.quote == address(0), "Wrong quote token");
        require(
            block.timestamp >= config.launch,
            "Ticket sales is not started"
        );
        require(
            _nTickets > 0,
            "Number of tickets for purchasing should be greater than zero"
        );
        require(msg.value > 0, "Payment should be greater than zero");

        if (depositList[msg.sender] == 0) {
            require(msg.value >= config.price, "Under least staking amount");
```

```
            require(
                msg.value <= config.price.mul(config.maxTicket),
                "Can't buy more than maximum per wallet"
            );
            require(
                config.price.mul(_nTickets) == msg.value,
                "Wrong number of tickets to purchase"
            );
        } else {
            revert("only once can buy tickets");
        }
        ...
    }

    function buyTicketQuote(uint256 _payment, uint256 _nTickets)
        public
        initialized
        isNotAllocated
        nonReentrant
    {
        require(config.quote != address(0), "Wrong quote token");
        require(
            block.timestamp >= config.launch,
            "Ticket sales is not started"
        );
        require(
            _nTickets > 0,
            "Number of tickets for purchasing should be greater than zero"
        );
        require(_payment > 0, "Payment should be greater than zero");

        if (depositList[msg.sender] == 0) {
            require(_payment >= config.price, "Under least staking amount");
            require(
                _payment <= config.price.mul(config.maxTicket),
                "Can't buy more than maximum per wallet"
            );
            require(
                config.price.mul(_nTickets) == _payment,
                "Wrong number of tickets to purchase"
            );
        }
        ...
    }
```

3.In the Subscription contract, the unStaking function has an `isNotAllocated` modifier but it also has the `require(allocStatus == false, "Can cancel subscription when NFTs are not allocated");` as a

redundant check to consume more gas in the contract.

Code location:

Subscription.sol#457-460

```
    function unStaking(uint256 _amount) public isNotAllocated nonReentrant {
        require(_amount > 0, "Unstake more the zero");
        require(
            allocStatus == false,
            "Can cancel subscription when NFTs are not allocated"
        );
       ......
    }
```

Solution

It's recommended to remove the redundant require check.

Status

Fixed

[N8] [Low] Missing check return value

Category: Others

Content

In these contracts, the return value of the transfer and transferFrom functions are not checked may lead to logical errors.

Code location:

Collection.sol#250, 334, 342

EvenAllocation.sol#557, 565

MysteryBox.sol#492, 558, 566

Subscription.sol#636, 644, 695, 703

```
        IERC20(quote).transferFrom(msg.sender, address(this), _payment);
        IERC20(quote).transfer(treasury, fee);
        IERC20(quote).transfer(payment, profit);
        IERC20(config.quote).transfer(config.treasury, share);
        IERC20(config.quote).transfer(config.payment, profit);
```

```
        IERC20(quote).transferFrom(msg.sender, address(this), _payment);
        IERC20(config.quote).transfer(to, _amount.sub(fee));
```

Solution

It is recommended to check the return value or use the SafeERC20.

Status

Fixed

[N9] [Low] Using the transfer function to transfer ETH may cause assets to be locked

Category: Others

Content

Transfer function uses a fixed gas limit (2300). If the recipient is a contract and implements a fallback function that uses more than 2300 gas, then the user can suffer from the lock of funds.

The transfer() and send() functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase in the SLOAD instruction.

Code location:

EvenAllocation.sol#455-461

MysteryBox.sol#396-402

Subscription.sol#551-557

```
    function requestRandomNumber() external payable {
        latestRandomizingBlock = block.number;
        uint256 _usedFunds = witnet.randomize{value: msg.value}();
        if (_usedFunds < msg.value) {
            payable(msg.sender).transfer(msg.value - _usedFunds);
        }
    }
```

Solution

It is recommended to use call() instead of transfer(), but be sure to follow the Checks-Effects-Interactions

principle or add reentrancy guards, and the return value should be checked.

Status

Fixed

[N10] [Suggestion] Missing the checks

Category: Others

Content

1.In the AirDrop, Collection, and MysteryBox contracts, the owner can modify some important parameters, and these functions lack the 0 value check.

Code location:

AirDrop.sol#106-114

Collection.sol#95-103

```
function setLaunch(uint256 _launch) public onlyOwner {
    launch = _launch;
    emit SetLaunch(launch);
}

function setLockup(uint256 _lockup) public onlyOwner {
    lockup = _lockup;
    emit SetLockup(lockup);
}
```

2.In the Collection contract, the owner role can modify the uris, amounts, and prices through the registerItems function, but it only checks the `require(uris.length == amounts.length, "Array length mismatch");`.

Code location:

Collection.sol#139-156

```
function registerItems(
    string[] memory uris,
    uint256[] memory amounts,
    uint256[] memory prices
) public onlyRole(MINTER_ROLE) isNotRegistered {
    require(uris.length == amounts.length, "Array length mismatch");
    ......
}
```

Solution

1. It is recommended to add the 0 address or 0 value check when sensitive parameters are modified.
2. It's recommended to check the three array lengths are equal.

Status

Fixed

[N11] [Suggestion] Gas Optimization

Category: Gas Optimization Audit

Content

1.In the contract, using assert will consume the remaining gas when the transaction fails to execute.

Code location:

EvenAllocation.sol#144, 464

MysteryBox.sol#127, 405

Subscription.sol#147, 560

```
        assert(address(_witnetRandomness) != address(0));
        assert(latestRandomizingBlock > 0);
```

2.In the EvenAllocation and Subscription contracts, A booking array will be set up in the contract, and the user's data will be pushed into this number after each buyTicketEth/buyTicketQuote and stakingEth/stakingQuote. Control the array length to prevent exceeding the stack depth. The array length is limited by the available stack depth and available gas. If the length of the array is too large, it may cause a stack overflow error or transaction execution failure.

Code location:

EvenAllocation.sol#162-297

Subscription.sol#172-291

```
    Book[] public booking;

        booking.push(
            Book({
                user: msg.sender,
```

```
                nTickets: _nTickets,
                deposit: _payment,
                allocated: 0,
                status: false,
                claimed: false
            })
        );

        booking.push(
                Book({
                    user: msg.sender,
                    staking: _payment,
                    balance: _payment,
                    rateMaxAlloc: max,
                    evenMaxAlloc: 0,
                    rateAlloc: 0,
                    evenAlloc: 0,
                    totalAlloc: 0,
                    status: false,
                    claimed: false
                })
            );
```

Solution

1. It is recommended to use require instead of assert to optimize gas.
2. It is recommended to limit the length of the array to prevent stack overflow error or transaction execution failure.

Status

Fixed

[N12] [Suggestion] Redundant code

Category: Others

Content

1.There are redundant codes in the EvenAllocation, MysteryBox, and Subscription contract.

Code location:

EvenAllocation.sol#473-482

MysteryBox.sol#411-418, 617-628

Subscription.sol#569-578

```
function _getIndexByAddress(address user) private view returns (uint256) {
    ......
}

function _deleteItemAfterMint(uint256 target) private {
    .......
}

function setMandatory(address[] calldata _nfts, bool _andOr)
    public
    onlyOwner
{
    ......
}

function _getIndexByAddress(address user) private view returns (uint256) {
    ......
}
```

2.After removing the setMandatory function, the `flagMandatory` parameter is initialized as false and can not be changed. It makes the `if (flagMandatory) {...}` judgment as redundant code in the claim function, and it will cause more gas for users to call the claim function.

Code location:

AirDrop.sol#196-211

```
function claim() public {
    require(block.timestamp > launch, "Not yet launch time reached");
    require(totalItems > totalSold, "aridrop sold out");

    if (flagMandatory) {
        ......
    }
```

3.In the EvenAllocation and Subscription contracts, the `isNotAllocated` and `allocated` modifiers are redundant code because there is no parameter change for the `allocStatus`.

Code location:

EvenAllocation.sol#100-108

```
    modifier allocated() {
        require(allocStatus == true, "Not yet mystery box allocated");
        _;
    }

    modifier isNotAllocated() {
        require(allocStatus == false, "Mystery box is already allocated");
        _;
    }
```

Solution

It is recommended to clarify the business involved. If this part of the logic is not needed, the code can be deleted to save gas, and at the same time, ensure that the set parameters are useful.

Status

Fixed

[N13] [Suggestion] Failure to follow the Checks-Effects-Interactions principle

Category: Replay Vulnerability

Content

In the AirDrop and Collection contracts, users can claim and buy ERC721 Tokens through the claim, buyItemCredit, buyItemQuote, and buyItemEth functions. However, during the claim or buy process, it first mints the NFT tokens to the user through the mintByTokenId function and then changes _tokenIdCounter, totalSold, and itemSolds values, which do not comply with the Checks-Effects-Interactions principle and without the reentrancy guards.

Code location:

AirDrop.sol#183-219, 221-263

Collection.sol#158-204, 206-267

```
            tokenId = tokenId.add(_tokenIdCounter[_itemIx].current());
            mintByTokenId(msg.sender, tokenId, itemURIs[_itemIx]);
            _tokenIdCounter[_itemIx].increment();
            itemSolds[_itemIx] = itemSolds[_itemIx].add(1);
```

```
        totalSold = totalSold.add(1);
       ...
        for (uint256 i = 0; i < _amount; i++) {
            uint256 tokenId = startId.add(_tokenIdCounter[_itemIx].current());
            mintByTokenId(msg.sender, tokenId, itemURIs[_itemIx]);
            _tokenIdCounter[_itemIx].increment();
        }

        settlement(_itemIx, _amount);
        itemSolds[_itemIx] = itemSolds[_itemIx].add(_amount);
```

Solution

It is recommended to follow the Checks-Effects-Interactions principle and change the status first before performing the transfer operation.

Status

Fixed

[N14] [Suggestion] Random number check

Category: Others

Content

In the MysteryBox contract, users can call the claim function to claim their NFT and the minter role can mint the NFT by the buyItemCredit function. The NFT tokenIds are calculated by an external Oracle from the _fetchRandomNumber function. There may exist a situation in which the calculated random number is the same as the one before. But the NFT tokenId can not be the same. The call will revert.

Code location:

MysteryBox.sol#332-369

```
        for (uint256 i = 0; i < amount; i++) {
            ......

            _fetchRandomNumber(uint32(totalItems));

            mintByClaim(to, getTokenURI(randomness), true);
        }
    }
```

Solution

It's recommended to check the random number whether is the same as the minted tokenId before.

Status

Fixed

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002311240002	SlowMist Security Team	2023.11.15 - 2023.11.24	Passed

Summary conclusion: The SlowMist security team use a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 2 high risk, 3 medium risk, 4 low risk, 5 suggestion vulnerabilities. All the findings were fixed. The code was not deployed to the mainnet.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist