Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.04.15, the SlowMist security team received the Ultiverse team's security audit application for Ultiverse - Chips Contract, 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

This is the Ultiverse Chip project. The GoldChip is an ERC721 NFT collection contract in which users can burn their NFTs to unlock ERC20 tokens. And users can mint the GoldChip NFT with the centrally signed MerkleProof in the ChipLaunch contract. For the ChipStakingPool contract, users can deposit native tokens to get entries to win GoldChip NFTs. There are three types of entries: Gold, Silver, and Public, which represent three different chances to obtain the qualification to mint GoldChip NFTs. All entries which do not win the mint qualification will be refunded, depositors can withdraw the refund verified by MerkleTree.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Medium Risk of excessive authority	Authority Control Vulnerability Audit	Medium	Acknowledged
N2	Low-level call reminder	Others	Information	Acknowledged
N3	Potential overflow risks caused by type conversion	Arithmetic Accuracy Deviation Vulnerability	Suggestion	Fixed
N4	Refund lock reminder	Design Logic Audit	Information	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/ultiverse-io/ChipsContract

commit: 83319b0eece6c0cd73fa006f1a7cfba2f93a170f

Fixed Version:

https://github.com/ultiverse-io/ChipsContract

commit: 5259de76c5f959e278ea21bfcebdfeb83612177c

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:

ChipLaunch			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable

ChipLaunch			
pause	External	Can Modify State	onlyOwner
unpause	External	Can Modify State	onlyOwner
setMerkleTreeRoot	External	Can Modify State	onlyOwner
mint	External	Can Modify State	whenNotPaused

ChipVesting			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
emergencyWithdraw	External	Can Modify State	onlyOwner
setVestingConfig	External	Can Modify State	onlyOwner
chipVestingSchedule	Public	-	-
claimVestedAmount	Public	Can Modify State	-
chipClaimableAmount	Public	-	-
_initialReleaseAfterBurn	Internal	Can Modify State	-

GoldChip			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC721
grantRole	Public	Can Modify State	onlyOwner
revokeRole	Public	Can Modify State	onlyOwner
setBurnable	Public	Can Modify State	onlyOwner
setBaseTokenURI	External	Can Modify State	onlyOwner
mint	External	Can Modify State	nonReentrant onlyRole

GoldChip			
burn	Public	Can Modify State	whenBurnable
_baseURI	Internal	-	-
supportsInterface	Public	-	-
_update	Internal	Can Modify State	-

ChipStakingPool			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
pause	External	Can Modify State	onlyOwner
unpause	External	Can Modify State	onlyOwner
setPoolConfig	External	Can Modify State	onlyOwner
setRefundConfig	External	Can Modify State	onlyOwner
withdraw	External	Can Modify State	onlyOwner
deposit	External	Payable	whenNotPaused
claimRefund	External	Can Modify State	whenNotPaused

4.3 Vulnerability Summary

[N1] [Medium] Medium Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

1.In the GoldChip contract, the owner role can modify key sensitive parameters such as the `burnable` status, the `_baseTokenURI`, and granting or revoking roles, which will lead to the risk of over-privilege of the owner role.

Code location:

src/Chip/GoldChip.sol#32-54

```
    function grantRole(
        bytes32 role,
        address account
    ) public override(AccessControl, IAccessControl) onlyOwner {
        _grantRole(role, account);
    }

    function revokeRole(
        bytes32 role,
        address account
    ) public override(AccessControl, IAccessControl) onlyOwner {
        _revokeRole(role, account);
    }

    function setBurnable(bool enabled) public onlyOwner {
        burnable = enabled;
        emit BurnableSet(enabled);
    }

    function setBaseTokenURI(string calldata uri) external onlyOwner {
        _baseTokenURI = uri;
        emit BaseTokenURIUpdated(uri);
    }
```

2.In the ChipLaunch, ChipVesting, and ChipStakingPool contracts, the owner role can modify key sensitive parameters such as the merkleTreeRoot, the vestingConfig, the poolConfig, the refundConfig, and withdrawing all `vestingConfig.erc20Contract` funds from ChipVesting contract and native tokens from ChipStakingPool contract, which will lead to the risk of over-privilege of the owner role.

Code location:

src/Chip/ChipLaunch.sol#34-37

src/Chip/ChipVesting.sol#60-70

src/Pool/ChipStakingPool.sol#32-49

```
    function setMerkleTreeRoot(bytes32 root) external onlyOwner {
        merkleTreeRoot = root;
        emit MerkleTreeRootSet(root);
    }
```

```
    function emergencyWithdraw(uint256 amount) external onlyOwner {
        bool transferResult =
IERC20(vestingConfig.erc20Contract).transfer(_msgSender(), amount);
        if (!transferResult) {
            revert FailedToTransfer();
        }
    }

    function setVestingConfig(VestingConfig memory config) external onlyOwner {
        vestingConfig = config;
        emit VestingConfigSet(config);
    }

    function setPoolConfig(PoolConfig calldata config) external onlyOwner {
        poolConfig = config;
        emit PoolConfigUpdated(config);
    }

    function setRefundConfig(RefundConfig calldata config) external onlyOwner {
        refundConfig = config;
        emit RefundConfigUpdated(config);
    }

    function withdraw(uint256 amount) external onlyOwner {
        emit BalanceWithdrawn(_msgSender(), amount);

        (bool success, ) = _msgSender().call{value: amount}("");
        if (!success) {
            revert FailedToTransfer();
        }
    }
```

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. 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

Acknowledged; After communicating with the project team, they expressed that they will use a multisig wallet for ownership. The project party restricted the owner withdraw method of chipcontract in the fixed version,

allowing the team to immediately withdraw the income part (2000ether). If there are remaining uncollected funds, it will take 3 months (2024-08-02 12:00:00 UTC) to open withdraw.

[N2] [Information] Low-level call reminder

Category: Others

Content

In the ChipStakingPool contract, the contract uses low-level calls and does not limit the amount of gas used to transfer native tokens to users.

Code location:

src/Pool/ChipStakingPool.sol#45, 163

```
        (bool success, ) = _msgSender().call{value: amount}("");
        (bool success, ) = _msgSender().call{value: refundAmount}("");
```

Solution

When using low-level calls, it is recommended to limit the amount of gas used.

Status

Acknowledged

[N3] [Suggestion] Potential overflow risks caused by type conversion

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the ChipStakingPool contract, users can deposit the native tokens to gain the Lots. The deposit function uses a type conversion to convert the uint256 type values such as quantity, availableGoldLots, and availableSilverLots to the uint64 type. If the user passes in a `quantity` greater than uint64, this will cause the overflow when converting the data to uint256.

Code location:

src/Pool/ChipStakingPool.sol#51-131

```
    function deposit(
        ...
    ) external payable whenNotPaused {
```

```
        ...
        uint256 availableGoldLots = goldLots - stats.goldLotEntryCount;
        uint256 availableSilverLots = silverLots - stats.silverLotEntryCount;
        uint64 newGoldEntryCount = 0;
        uint64 newSilverLastEntryIndex = 0;
        uint64 newPublicLastEntryIndex = 0;
        ...
            if (availableSilverLots >= quantity) {
                stats.silverLotEntryCount += uint64(quantity);
                quantity = 0;
                newSilverLastEntryIndex = lastEntryIndex + uint64(quantity);
            } else {
                stats.silverLotEntryCount += uint64(availableSilverLots);
                quantity -= availableSilverLots;
                newSilverLastEntryIndex = lastEntryIndex +
uint64(availableSilverLots);
            }
            ...
            stats.publicEntryCount += uint64(quantity);
            newPublicLastEntryIndex = lastEntryIndex + uint64(quantity);
           ...
    }
```

Solution

It's recommended to use the OpenZeppelin's SafeCast for type conversion.

Status

Fixed

[N4] [Information] Refund lock reminder

Category: Design Logic Audit

Content

In the ChipStakingPool contract, users can deposit native tokens to get Lots and claim refunds to get the native back. When calling these two functions, these two functions will check the MerkleProof signed by the central. The claimRefund function will also check whether the user claimed before by checking the `userRefundCount[_msgSender()]` is larger than 0. If the signed MerkleProof quantity is not the same as the user's deposit. There will be a situation where the user will not be able to call the claimRefund function again to withdraw native tokens after claiming refunds once.

Code location:

src/Pool/ChipStakingPool.sol#133-167

```
    function claimRefund(uint256 quantity, bytes32[] calldata proof) external
whenNotPaused {
        ...
        if (!MerkleProof.verify(proof, refundConfig.merkleTreeRoot, leaf)) {
            revert InvalidProof();
        }
        if (userRefundCount[_msgSender()] > 0) {
            revert UserAlreadyRefunded();
        }
        UserStats storage stats = userStats[_msgSender()];
        uint256 userEntries = stats.goldLotEntryCount +
            stats.silverLotEntryCount +
            stats.publicEntryCount;
        if (quantity > userEntries) {
            revert InsufficientEntriesToRefund();
        }
        userRefundCount[_msgSender()] = quantity;
        uint256 refundAmount = quantity * poolConfig.chipPrice;
        emit RefundClaimed(_msgSender(), quantity, refundAmount);
        (bool success, ) = _msgSender().call{value: refundAmount}("");
        if (!success) {
            revert FailedToTransfer();
        }
    }
```

Solution

After communicating with the project team, they expressed that not all the `userEntries` will be refunded, only the entries which do not win the GoldChip NFT mint qualification can be refunded. If an Entry wins, then the ETH deposited by that entry will be paid to the team. So the `quantity` is not greater than the `userEntries`.

Status

Acknowledged

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002404170002	SlowMist Security Team	2024.04.15 - 2024.04.17	Medium Risk

Summary conclusion: The SlowMist security team uses a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 1 medium risk, 1 suggestion, and 2 information. 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