Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.07.29, the SlowMist security team received the Match team's security audit application for Match, 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

Match is an innovative platform committed to breaking down barriers to value-creating social interactions. Based on AI and bottom layer of the big data architecture, this platform leverages the wealth effect of meme coins to establish an efficient value-creating social network. On this platform, projects are accurately matched with users based on extensible social graphs, so that social interactions among users can be combined with wealth.

This is an audit of the contracts for the Match, which mainly includes the Auction, Deposits, Pool, Token, Airdrop, and Token Staking modules. The Token module is primarily used for issuing NFTs and RFG tokens. It is important to note that there is an upper limit on the token issuance; whitelisted users can claim token airdrops through the Airdrop module; users can also stake specific assets in the Token Staking module to obtain RFG token rewards; the Auction module allows users to participate in NFT auctions using USDC; users can also deposit RFG tokens in the Deposits

module, and while earning RFG token rewards, they can also increase their boost to improve their chances of obtaining high-value NFTs when participating in auctions. Users can stake NFTs through the Pool module to earn RFG token rewards.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Potentially unclaimed rewards	Arithmetic Accuracy Deviation Vulnerability	Low	Acknowledged
N2	Optimizable refreshGlobalState	Gas Optimization Audit	Suggestion	Fixed
N3	Compatibility issues with deflationary tokens	Design Logic Audit	Medium	Acknowledged
N4	Optimizable RFG token distribution method	Design Logic Audit	Suggestion	Acknowledged
N5	Admin who has not set the Boss role	Authority Control Vulnerability Audit	Low	Acknowledged
N6	Not checking the reasonableness of time when updating auctions	Design Logic Audit	High	Fixed
N7	Smart contracts cannot participate in the auction	Design Logic Audit	Information	Acknowledged
N8	Time check when closing auction is flawed	Design Logic Audit	Medium	Fixed
N9	Not checking if the user's bid is as expected	Design Logic Audit	Low	Fixed
N10	Not checking if the user's bid is refundable	Design Logic Audit	Low	Fixed

NO	Title	Category	Level	Status
N11	Risk of pseudo-randomness	Block data Dependence Vulnerability	Critical	Fixed
N12	Redundant return value of adjustRandomtoken function	Others	Suggestion	Acknowledged
N13	The tokenId obtained by the user is related to the NFT inventory	Design Logic Audit	Information	Acknowledged
N14	Unchecked boost bound parameters during initialization	Design Logic Audit	Suggestion	Fixed
N15	Rewards not settled as expected	Design Logic Audit	Critical	Fixed
N16	Incorrect whitelist pool check	Unsafe External Call Audit	Critical	Fixed
N17	When `totalStakedItem` is 0, the reward should be returned directly as 0	Design Logic Audit	Suggestion	Fixed
N18	Unexpected rewards when staking in pairs	Design Logic Audit	Critical	Fixed
N19	Manipulate boost to influence the token id obtained in the auction	Design Logic Audit	Medium	Acknowledged
N20	Redundant PoolMax enum	Others	Suggestion	Fixed
N21	Reward calculation for two stakers in GroupPool being the same user	Design Logic Audit	Information	Acknowledged
N22	Potentially incorrect social staking reward information in GroupPool	Design Logic Audit	Low	Fixed
N23	Potential Denial of Service Risk	Denial of Service Vulnerability	Low	Acknowledged

NO	Title	Category	Level	Status
N24	The validity of the pid was not checked when creating the pool	Design Logic Audit	Suggestion	Fixed
N25	When creating a matchCode, it does not check whether the pool has been created.	Design Logic Audit	Suggestion	Fixed
N26	There is an upper limit on the matchCodes available in the pool	Design Logic Audit	Suggestion	Fixed
N27	Checks-Effects-Interactions are not followed when transferring out NFT	Design Logic Audit	Suggestion	Confirmed
N28	Optimizable reward information update	Gas Optimization Audit	Suggestion	Fixed
N29	Risks of excessive privilege	Authority Control Vulnerability Audit	Medium	Acknowledged
N30	Protocol Missing Emergency Operations Role	Authority Control Vulnerability Audit	Suggestion	Fixed
N31	Missing event records	Others	Suggestion	Fixed

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/socialmatch/ssm-contract

commit: 8d2ca560d31ffff27b0d2281a87c7a63f7fdafb6

Fixed Version:

https://github.com/socialmatch/ssm-contract

commit: dc3b2f4aa105a7bad197194c58e3e5cafbb0357d

Audit Scope:

- contracts/Airdrop.sol
- contracts/SingleTokenStaking.sol
- contracts/auction/Auction.sol
- contracts/auction/libauction.sol
- contracts/deposites/*.sol
- contracts/smspool/*.sol
- contracts/tokens/*.sol

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:

Airdrop			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable EIP712
<Receive Ether>	External	Payable	-
withdrawToken	External	Can Modify State	onlyOwner
setRfgToken	External	Can Modify State	onlyOwner
setSigner	External	Can Modify State	onlyOwner
claimAirdrop	External	Can Modify State	-

InviteReward			
Function Name	Visibility	Mutability	Modifiers

InviteReward			
<Constructor>	Public	Can Modify State	Ownable EIP712
<Receive Ether>	External	Payable	-
withdrawToken	External	Can Modify State	onlyOwner
setRfgToken	External	Can Modify State	onlyOwner
setSigner	External	Can Modify State	onlyOwner
claimReward	External	Can Modify State	-

SingleTokenStaking			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
initialize	External	Can Modify State	initializer
_authorizeUpgrade	Internal	Can Modify State	onlyOwner
resetRewardSpeed	External	Can Modify State	onlyOwner
refreshGlobalState	Internal	Can Modify State	-
addDeposit	Internal	Can Modify State	-
subDesposit	Internal	Can Modify State	-
deposit	External	Can Modify State	-
withdraw	External	Can Modify State	-
claimReward	External	Can Modify State	-
getPendingReward	External	-	-

NftCard			
Function Name	Visibility	Mutability	Modifiers

NftCard			
<Constructor>	Public	Can Modify State	ERC1155 Ownable
setMinter	External	Can Modify State	onlyOwner
initIssue	External	Can Modify State	onlyOwner
mint	External	Can Modify State	onlyMinter
setUri	External	Can Modify State	onlyOwner
uri	Public	-	-

RFGToken			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC20 Ownable
setMinter	External	Can Modify State	onlyOwner
claimAirdrop	External	Can Modify State	onlyOwner
claimLiquidity	External	Can Modify State	onlyOwner
mint	External	Can Modify State	onlyMinters

Auction			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	External	Can Modify State	initializer
_authorizeUpgrade	Internal	Can Modify State	onlyRole
setMaxBidableTimes	External	Can Modify State	onlyRole
setOperator	External	Can Modify State	onlyRole
createAuction	External	Can Modify State	onlyRole

Auction			
updateAuction	External	Can Modify State	onlyRole
bidAuction	External	Can Modify State	-
finishAuction	External	Can Modify State	onlyRole
claimNft	External	Can Modify State	-
refund	External	Can Modify State	-
withdraw	External	Can Modify State	onlyRole
transferUsdtIn	Internal	Can Modify State	-
transferUsdtOut	Internal	Can Modify State	-
doMintNft	Internal	Can Modify State	-
doMintOneNft	Internal	Can Modify State	-
fakeRandomToken	Internal	-	-
adjustRandomtoken	Internal	Can Modify State	-

RFGDeposit			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
initialize	External	Can Modify State	initializer
_authorizeUpgrade	Internal	Can Modify State	onlyOwner
stopFixedPool	External	Can Modify State	onlyOwner
setFlexibleRewardSpeed	External	Can Modify State	onlyOwner
setStakingManager	External	Can Modify State	onlyOwner
getDepositBoost	External	-	-
updateBoostApy	Internal	Can Modify State	-

RFGDeposit			
calculateBoost	Internal	-	-
updateFlexibleBoost	Internal	Can Modify State	-
recalculateBoostApy	Internal	Can Modify State	-
flexibleDeposit	External	Can Modify State	-
flexibleWithdraw	External	Can Modify State	-
claimFlexibleReward	External	Can Modify State	-
createFixedDeposit	External	Can Modify State	onlyOwner
fixedDeposit	External	Can Modify State	-
setAutoRedeposit	External	Can Modify State	-
fixedWithdraw	External	Can Modify State	-
transferRFGIn	Internal	Can Modify State	-
transferRFGOut	Internal	Can Modify State	-
mintReward	Internal	Can Modify State	-
getPendingFlexibleReward	External	-	-
getJoinedFixedDeposit	External	-	-
estimateBoost	External	-	-

Pool			
Function Name	Visibility	Mutability	Modifiers
implementation	External	-	-
_authorizeUpgrade	Internal	Can Modify State	onlyManager
setRewardSpeed	External	Can Modify State	onlyManager
nextMatchCode	External	Can Modify State	onlyManager

Pool			
getNextMatchCode	External	-	-
getPool	Internal	-	-
assertCallerIsPool	Internal	-	-
selfStake	External	Can Modify State	onlyManager
selfUnstake	External	Can Modify State	onlyManager
calculateStakeReward	Internal	-	-
claimSelfStakeReward	Public	Can Modify State	onlyManager
getSelfStakePendingReward	Public	-	-
claimAllCustodialReward	Public	Can Modify State	-
claimCustodialReward	Public	Can Modify State	-
claimAllSocialReward	Public	Can Modify State	-
claimSocialReward	Public	Can Modify State	-
socialStake	External	Can Modify State	-
cancelSocialStake	External	Can Modify State	-
socialUnstake	External	Can Modify State	-
forceSocialUnstake	External	Can Modify State	-
custodialNft1	External	Can Modify State	-
uncustodialNft1	External	Can Modify State	-
custodialNft2	External	Can Modify State	-
uncustodialNft2	External	Can Modify State	-
getAllCustodialPendingReward	Public	-	-
getCustodialPendingReward	Public	-	-

Pool			
getAllSocialPendingReward	Public	-	-
getSocialPendingReward	Public	-	-

SinglePool			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
custodialNft1	Public	Can Modify State	-
uncustodialNft1	Public	Can Modify State	-
updateCustodialInfo	Internal	Can Modify State	-
claimAllReward	Public	Can Modify State	onlyManager
claimAllCustodialReward	Public	Can Modify State	onlyManager
claimCustodialReward	Public	Can Modify State	onlyManager
getPendingReward	External	-	-
getAllCustodialPendingReward	Public	-	-
getCustodialPendingReward	Public	-	-
getCustodialMatchCodes	External	-	-

StakingManager			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
_authorizeUpgrade	Internal	Can Modify State	onlyRole
initialize	Public	Can Modify State	initializer

StakingManager			
upgradePool	External	Can Modify State	onlyRole
createPool	External	Can Modify State	onlyRole
selfStake	External	Can Modify State	onlyValidPool
selfUnstake	External	Can Modify State	onlyValidPool
createMatch2Code	External	Can Modify State	-
createMatch3Code	External	Can Modify State	-
stakeNft	Internal	Can Modify State	-
socialStake	External	Can Modify State	-
cancelSocialStake	External	Can Modify State	-
applySocialUnstake	External	Can Modify State	-
agreeSocialUnstake	External	Can Modify State	-
forceSocialUnstake	External	Can Modify State	-
claimAllSocialReward	External	Can Modify State	-
claimSocialReward	Public	Can Modify State	-
claimAllCustodialReward	External	Can Modify State	-
claimCustodialReward	External	Can Modify State	-
claimSelfStakeReward	External	Can Modify State	-
claimPoolReward	External	Can Modify State	-
claimAllPoolReward	External	Can Modify State	-
getAllPendingRewards	External	-	-
getMatchInfo	External	-	-
onERC1155Received	External	-	-

StakingManager			
returnNftsBackAndClaimReward	Internal	Can Modify State	-
transferInNft	Internal	Can Modify State	-
transferOutNft	Internal	Can Modify State	-
mintCustodialReward	Internal	Can Modify State	-
doMintRFGWithBoost	Internal	Can Modify State	-
pid2tokens	Internal	-	-

CouplePool			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	External	Can Modify State	initializer
socialStake	External	Can Modify State	onlyManager
cancelSocialStake	External	Can Modify State	onlyManager
socialUnstake	External	Can Modify State	onlyManager
forceSocialUnstake	External	Can Modify State	onlyManager
custodialNft2	Public	Can Modify State	-
uncustodialNft2	Public	Can Modify State	-
updateCustodialInfo	Internal	Can Modify State	-
updateSocialStakeRewardInfo	Internal	Can Modify State	-
updateCustodialRewardInfo	Internal	Can Modify State	-
claimAllReward	External	Can Modify State	onlyManager
claimAllCustodialReward	Public	Can Modify State	onlyManager
claimCustodialReward	Public	Can Modify State	onlyManager

CouplePool			
claimAllSocialReward	Public	Can Modify State	onlyManager
claimSocialReward	Public	Can Modify State	onlyManager
viewSocialStakeRewardInfo	Internal	-	-
viewCustodialRewardInfo	Internal	-	-
getPendingReward	External	-	-
getAllSocialPendingReward	Public	-	-
getSocialPendingReward	Public	-	-
getAllCustodialPendingReward	Public	-	-
getCustodialPendingReward	Public	-	-
getCustodialMatchCodes	External	-	-
getJoinedMatchCodes	External	-	-

GroupPool			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	External	Can Modify State	initializer
socialStake	External	Can Modify State	onlyManager
cancelSocialStake	External	Can Modify State	onlyManager
socialUnstake	External	Can Modify State	onlyManager
forceSocialUnstake	External	Can Modify State	onlyManager
updateRewardInfo	Internal	Can Modify State	-
claimAllReward	External	Can Modify State	onlyManager
claimAllSocialReward	Public	Can Modify State	onlyManager

GroupPool			
claimSocialReward	Public	Can Modify State	onlyManager
viewSocialStakeRewardInfo	Internal	-	-
getPendingReward	External	-	-
getAllSocialPendingReward	Public	-	-
getSocialPendingReward	Public	-	-
getJoinedMatchCodes	External	-	-

4.3 Vulnerability Summary

[N1] [Low] Potentially unclaimed rewards

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the SingleTokenStaking contract, the calculation of staking rewards mainly depends on the rewardSpeed variable. The contract calculates the global reward accumulation accuedReward based on rewardSpeed and the block interval. When a user settles rewards, the difference between the current global accuedReward and the user's last settled accuedReward is multiplied by the user's deposit amount to determine the user's claimable rewards. It is important to note that the contract does not limit users' minimum deposit amount. This means that when a user deposits an extremely small amount and rewardSpeed is set relatively low (for example, if the user deposits 1 wei and rewardSpeed is less than 1e18), the user's small rewards may be truncated due to decimal rounding during the reward settlement process. This may result in the user's rewards being left unclaimed in the contract.

Code location:

contracts/SingleTokenStaking.sol#L79

contracts/SingleTokenStaking.sol#L149

```
    function refreshGlobalState() internal {
        DepositConfig memory config = gDeposits;
        if (config.totalDeposited != 0) {
            config.accuedReward += (block.number - config.accuedBlock) *
```

```
config.rewardSpeed * DivPrecision / config.totalDeposited;
        }
        ...
    }

    function claimReward() external {
        ...
        uint256 reward = (gDeposits.accuedReward - userDeposit.accuedReward) *
userDeposit.amount / DivPrecision;
        ...
    }
```

Solution

It is recommended that the minimum deposit amount be limited for users or that the value of rewardSpeed be carefully considered when setting it.

Status

Acknowledged

[N2] [Suggestion] Optimizable refreshGlobalState

Category: Gas Optimization Audit

Content

In the SingleTokenStaking contract, the refreshGlobalState function is used to update the global reward state accuedReward and update the accumulated rewards and the current block to the corresponding global variables. It is important to note that there may be a large number of users performing operations such as depositing, withdrawing, and claiming rewards within the same block, which will result in frequent calls to the refreshGlobalState function. This means that although accuedReward will not be accumulated within the same block, users still need to pay some gas to update `gDeposits.accuedReward` and `gDeposits.accuedBlock`, which is unnecessary within the same block.

Code location: contracts/SingleTokenStaking.sol#L79

```
    function refreshGlobalState() internal {
        DepositConfig memory config = gDeposits;
        if (config.totalDeposited != 0) {
            config.accuedReward += (block.number - config.accuedBlock) *
config.rewardSpeed * DivPrecision / config.totalDeposited;
        }
```

```
        gDeposits.accuedReward = config.accuedReward;
        gDeposits.accuedBlock = block.number;
    }
```

Solution

It is recommended to check in refreshGlobalState whether `gDeposits.accuedBlock` is equal to the current block, and if it is, directly return to save gas.

Status

Fixed

[N3] [Medium] Compatibility issues with deflationary tokens

Category: Design Logic Audit

Content

In the SingleTokenStaking contract, users can deposit supported tokens into the contract using the deposit function, and the addDeposit function directly records the amount of deposit tokens passed in by the user. If the token supported by the contract is deflationary, the contract will actually receive fewer tokens than the deposit amount passed in by the user. This will cause the contract to record a higher user deposit than the actual amount of tokens received. When the user withdraws, it will result in a bad debt for the protocol.

Code location: contracts/SingleTokenStaking.sol#L118

```
    function deposit(uint256 amount) external {
        require(amount != 0, "invalid amount");

        SafeERC20.safeTransferFrom(IERC20(gDeposits.tokenAddress), msg.sender,
address(this), amount);

        ...
    }
```

Solution

If the SingleTokenStaking contract supports the deposit of deflationary tokens, then the difference between the contract balance before and after the user's deposit should be recorded as the user's actual deposit amount.

Status

Acknowledged; After communicating with the project team, the project team stated that the protocol will not support any deflationary tokens.

[N4] [Suggestion] Optimizable RFG token distribution method

Category: Design Logic Audit

Content

In the RFGToken contract, the token distribution rules are hardcoded. 30% of the token supply will be allocated to liquidity, 10% will be allocated to airdrops, and the remaining tokens will be minted by the minter role. The contract uses three separate functions to mint tokens for these three different allocation purposes. However, it should be noted that in the claimAirdrop and claimLiquidity functions, although the tokens are minted for airdrop and liquidity purposes, the receiving addresses are not specified. The owner role can mint these tokens to any address.

Code location: contracts/tokens/RFGToken.sol#L38-L56

```
    function claimAirdrop(address to, uint256 value) external onlyOwner {
        ...

        _mint(to, value);

        emit Airdroped(to, value);
    }

    function claimLiquidity(address to, uint256 value) external onlyOwner {
        ...

        _mint(to, value);

        emit LiquidityClaimed(to, value);
    }
```

Solution

If the token contract has already clearly specified the airdrop contract and liquidity receiving addresses at the time of deployment, it is recommended to mint these tokens to fixed addresses to enhance community trust.

Status

Acknowledged

[N5] [Low] Admin who has not set the Boss role

Category: Authority Control Vulnerability Audit

Content

During the initialization of the Auction contract, the specified boss address is granted the BossRole. However, it should be noted that the BossRole is not assigned the AdminRole. This means that if the boss address experiences issues such as private key leakage, the protocol will not be able to handle the boss address through `revokeRole/grantRole`.

Code location: contracts/auction/Auction.sol#L93

```
    function initialize(
        string memory name,
        string memory version,
        address admin,
        address nftCard,
        address[] calldata operators,
        address boss,
        address rfgDeposit
    )
        external
        initializer
    {
        ...
        _grantRole(BossRole, boss);
        ...
    }
```

Solution

If this is not an intended design, it is recommended to set the AdminRole for the BossRole during contract initialization.

Status

Acknowledged; After communicating with the project team, the project team stated that once the boss role is set, it will not be modified again, and the boss role is managed by a multisig wallet.

[N6] [High] Not checking the reasonableness of time when updating auctions

Category: Design Logic Audit

Content

In the Auction contract, the admin role can update existing auction configurations through the updateAuction function. When updating, it checks whether the new startPrice is greater than 0, but it does not check whether the new endTime is greater than startTime. It should be noted that the admin can update an auction that has already ended to reopen it. This means that users who have already placed bids or claimed items can participate in the auction again. However, this will cause the restarted auction to conflict with the previous claim/refund data. For example, if a user who successfully claimed an NFT in the previous auction wins the auction again, they will not be able to claim the new NFT successfully a second time. This does not align with the expected design.

Code location: contracts/auction/Auction.sol#L128-L135

```
    function updateAuction(uint8 aucId, AuctionSetting calldata auction) external
onlyRole(AdminRole) {
        require(auctions[aucId].endTime > 0, "auction not exist");
        require(auction.startPrice > 0, "invalid price");

        auctions[aucId] = auction;

        emit AuctionUpdated(aucId, auction);
    }
```

Solution

It is recommended to only allow updating auction configurations that are still within the auction cycle and to check that the new endTime must be greater than startTime.

Status

Fixed

[N7] [Information] Smart contracts cannot participate in the auction

Category: Design Logic Audit

Content

In the Auction contract, users can participate in the auction through the bidAuction function. However, the function checks whether `msg.sender` is equal to `tx.origin`, which prevents smart contracts (including EIP4337 wallets) from participating in the auction. It should be noted that in the future, if the EIP3074 standard is approved, it may break this check.

Code location: contracts/auction/Auction.sol#L156

```
    function bidAuction(uint8 aucId, uint256 usdtAmount, bytes memory signature)
external {
        ...
        require(tx.origin == msg.sender, "not for contract");
        ...
    }
```

Solution

N/A

Status

Acknowledged

[N8] [Medium] Time check when closing auction is flawed

Category: Design Logic Audit

Content

In the Auction contract, the operator can end the auction through the finishAuction function after the endTime. When performing the finishAuction operation, the endTime is checked using `block.timestamp >= auction.endTime`, while when performing the bidAuction operation, the endTime is checked using `block.timestamp <= auction.endTime`. This means that when the operator performs the finishAuction operation to set `result.price` exactly at the endTime, users can still perform the bidAuction operation to participate in the auction. This may not align with the intended design. It may also cause confusion for users, as they can place a bid higher than `result.price` at the endTime but are not included in the final Merkle tree.

Code location: contracts/auction/Auction.sol#L177

```
    function bidAuction(uint8 aucId, uint256 usdtAmount, bytes memory signature)
external {
        ...
        require(
            block.timestamp <= auction.endTime,
            "already finished"
        );
        ...
    }
```

```
    function finishAuction(uint8 aucId, bytes32 merkleRoot, uint256 totalSold,
uint256 price) external onlyRole(OperatorRole) {
        ...
        require(block.timestamp >= auction.endTime, "auction not finished");
        ...
    }
```

Solution

It is recommended to check that the current time must be greater than `auction.endTime` and cannot be equal to it when performing the finishAuction operation.

Status

Fixed

[N9] [Low] Not checking if the user's bid is as expected

Category: Design Logic Audit

Content

In the Auction contract, users who successfully win the auction can obtain the NFT through the claimNft function. The operator sets the Merkle proof to verify the validity of the claiming user. When the user's bid price is higher than `auctionResult.price`, the contract processes a refund for them. However, the contract does not check whether the user's bid price is necessarily greater than or equal to `auctionResult.price`. If the Merkle tree erroneously includes users with bid prices lower than `auctionResult.price`, it may result in insufficient funds in the contract for the boss role to withdraw.

Code location: contracts/auction/Auction.sol#L189

```
    function claimNft(uint8 aucId, uint256 nftAmount, uint256 seed, bytes32[] memory
proof) external {
        AuctionResult memory auctionResult = auctionResults[aucId];
        require(auctionResult.price > 0, "auction not finished");
        require(nftAmount == 1 || nftAmount == 2, "invalid nft amount");

        require(!claimedNfts[aucId][msg.sender], "have claimed");
        require(refunds[aucId][msg.sender] == 0, "have refunded");

        ...
    }
```

Solution

It is recommended to check in the claimNft function that the user's bid price must be greater than `auctionResult.price`.

Status

Fixed

[N10] [Low] Not checking if the user's bid is refundable

Category: Design Logic Audit

Content

In the Auction contract, users who meet the refund conditions can use the operator's signature to request a refund. Theoretically, if a user is eligible to claim the NFT, the operator will not sign for them to avoid giving up their eligibility for a refund. However, the refund function does not strictly check whether the bid prices of all refunding users are less than `auctionResult.price`. If the operator erroneously signs a refund for a user who is eligible to claim the NFT, it will prevent the boss from withdrawing the remaining auction proceeds.

Code location: contracts/auction/Auction.sol#L214

```
function refund(uint8 aucId, bytes memory signature) external {
    AuctionResult memory auctionResult = auctionResults[aucId];
    require(auctionResult.price > 0, "auction not finished");

    uint256 refundAmount = bids[aucId][msg.sender];
    require(refundAmount > 0, "not bid");

    require(!claimedNfts[aucId][msg.sender], "have claimed");
    require(refunds[aucId][msg.sender] == 0, "have refunded");

    ...
}
```

Solution

It is recommended to strictly check that the user's bid price must be less than `auctionResult.price` when performing the refund operation.

Status

Fixed

[N11] [Critical] Risk of pseudo-randomness

Category: Block data Dependence Vulnerability

Content

In the Auction contract, when a user claims an NFT, the fakeRandomToken function is used to calculate the tokenId for the user. The fakeRandomToken function uses `block.prevrandao`, `block.number`, and the user-provided seed for calculation. Unfortunately, these parameters can be controlled or are already known. This allows malicious users to ensure that the tokenIds of the NFTs they obtain at specific blocks are all of high value.

Code location: contracts/auction/Auction.sol#L297

```
function fakeRandomToken(address to, uint256 seed) internal view returns(uint8) {
        ...
        uint256 n = uint256(keccak256(abi.encode(block.prevrandao, block.number,
seed)));
        uint256 v = n % 100;
        if (v <= tokenARange) { // init 50%
            return TokenA;
        } else if (v <= tokenBRange) { //init 30%
            return TokenB;
        } else {
            return TokenC; // init 20%
        }
    }
```

Solution

Using Chainlink VRF is the best practice for using random numbers on-chain, but it comes with a higher cost. Another feasible solution is to determine a future block (e.g., 4 epochs ahead) when ending the auction. When that block is reached, `block.prevrandao` is obtained as a fixed seed. When a user claims an NFT, the fixed `block.prevrandao` and `msg.sender` are used to calculate the tokenId. Since `msg.sender` is already fixed at the end of the auction, and `block.prevrandao` is from a designated future block, it can better satisfy the randomness requirement.

Status

Fixed; The current mitigation solution involves the project team selecting any block 10 minutes after the auction ends

and using `block.prevrandao` as the random number seed to alleviate the aforementioned risk. It is important to note that this solution still leads to an excessive privilege risk.

[N12] [Suggestion] Redundant return value of adjustRandomtoken function

Category: Others

Content

In the Auction contract, the adjustRandomtoken function is used to select a matching id for the user based on the current tokenId inventory. When all the inventory has been claimed, the function directly throws an error using `require(false)`. This makes the final `return 0` redundant because the function will never execute this return statement.

Code location: contracts/auction/Auction.sol#L333

```
    function adjustRandomtoken(uint8 aucId, uint8 token) internal returns(uint8) {
        ...
        require(false, "no token to claim");
        return 0;
    }
```

Solution

It is recommended to remove the redundant `return 0`.

Status

Acknowledged

[N13] [Information] The tokenId obtained by the user is related to the NFT inventory

Category: Design Logic Audit

Content

In the Auction contract, the adjustRandomtoken function is used to adjust the final tokenId based on the inventory of each tokenId's NFTs. If a user obtains the highest-value NFT but there is no inventory for this NFT, they may be assigned the lowest-value NFT instead, and vice versa.

Code location: contracts/auction/Auction.sol#L308-L334

```
 function adjustRandomtoken(uint8 aucId, uint8 token) internal returns(uint8) {
        AuctionSetting memory auction = auctions[aucId];
        AuctionResult memory auctionResult = auctionResults[aucId];

        uint8 guardToken = token;
        do {
            if (token == TokenA && auctionResult.tokenAClaimed < auction.tokenAAmount)
{
                auctionResults[aucId].tokenAClaimed += 1;
                return TokenA;
            }

            if (token == TokenB && auctionResult.tokenBClaimed < auction.tokenBAmount)
{
                auctionResults[aucId].tokenBClaimed += 1;
                return TokenB;
            }

            if (token == TokenC && auctionResult.tokenCClaimed < auction.tokenCAmount)
{
                auctionResults[aucId].tokenCClaimed += 1;
                return TokenC;
            }

            token = (token << 1) % 7;
        } while (token != guardToken);

        require(false, "no token to claim");
        return 0;
    }
```

Solution

N/A

Status

Acknowledged; The project team said this was the intended design.

[N14] [Suggestion] Unchecked boost bound parameters during initialization

Category: Design Logic Audit

Content

In the initialize function of the RFGDeposit contract, when the proxy contract is initialized, parameters such as lowerBound and upperBound are passed in. However, the function does not check whether the passed-in

lowerBound is less than upperBound. Incorrectly passing the corresponding values may cause the protocol to be unusable.

Code location: contracts/deposites/RFGDeposit.sol#L59-L60

```
    function initialize(
        address owner,
        address rfgToken,
        uint256 flexibleRewardSpeed,
        uint256 lowerBound,
        uint256 upperBound,
        uint256 factor
    )
        external
        initializer
    {
        ...
    }
```

Solution

It is recommended to check whether lowerBound is less than upperBound during initialization.

Status

Fixed

[N15] [Critical] Rewards not settled as expected

Category: Design Logic Audit

Content

In the RFGDeposit contract, users can make fixed-term deposits through the fixedDeposit function. When a user's autoRedeposit status is false, even if the user's deposit time is several times longer than the duration, only one cycle of rewards will be settled for the user. Unfortunately, the fixedDeposit function does not handle the case where autoRedeposit is false. This allows users with autoRedeposit set to false to make a small deposit to the same fixId after a long deposit period and still receive the full rewards, not just for one duration.

Code location: contracts/deposites/RFGDeposit.sol#L275

```
function fixedDeposit(uint64 fixId, uint256 amount, bool autoRedeposit) external {
        ...
```

```
        if (fDeposit.startTime == 0) {
            ...
        } else {
            fDeposit.pendingReward  += fDeposit.apy * (block.timestamp -
fDeposit.startTime) * fDeposit.amount / (365 days * DivPrecision);
            fDeposit.amount         += amount;
            fDeposit.startTime      = block.timestamp;
            fDeposit.autoRedeposit  = autoRedeposit;
        }
        ...
    }
```

Solution

It is recommended to check the user's previous autoRedeposit status when making a fixed-term deposit to calculate the rewards accordingly.

Status

Fixed

[N16] [Critical] Incorrect whitelist pool check

Category: Unsafe External Call Audit

Content

In the Pool contract, the assertCallerIsPool function is used to check whether the passed-in sender is a pool created in the stakingManager. The assertCallerIsPool function receives the `msg.sender` from SinglePool and CouplePool as a possible pool address, calls the poolID interface of `msg.sender` to obtain the pool id, and finally checks whether this pool id is valid in the stakingManager. Unfortunately, this check method is not effective. Malicious contracts can also implement the poolID interface and return a valid pool id (1~7) when called. Since the assertCallerIsPool function only checks whether the id is valid through the stakingManager contract, malicious contracts can easily bypass this check to perform malicious custodial staking and eventually exhaust the protocol's assets.

Code location: contracts/smspool/Pool.sol#L61-L63

```
    function assertCallerIsPool(address sender) internal view returns(PoolID) {
        PoolID fromPoolID = IPool(sender).poolID();
        require(
            IStakingManager(stakingManager).pools(fromPoolID) != address(0),
```

```
            "from pool not exist"
        );
        return fromPoolID;
    }
```

Solution

It is recommended to add a poolList in the stakingManager contract to store newly created pools. The assertCallerIsPool function should check whether `msg.sender` is in the poolList of the stakingManager contract to determine if it is a valid pool.

Status

Fixed

[N17] [Suggestion] When `totalStakedItem` is 0, the reward should be returned directly as 0

Category: Design Logic Audit

Content

In the Pool contract, the getSelfStakePendingReward function is used to query the amount of rewards a staker can currently receive. Theoretically, when the contract's totalStakedItem is 0, no user should be able to receive rewards. However, the getSelfStakePendingReward function still calculates rewards when totalStakedItem is 0, which is not as expected.

The same is true for the getSocialPendingReward/getCustodialPendingReward functions in the CouplePool and GroupPool contracts.

Code location: contracts/smspool/Pool.sol#L122

```
function getSelfStakePendingReward(address staker) public view override
returns(uint256) {
        if (selfStakeInfos[staker].amount == 0) return 0;

        Reward.Info memory gr = gRewardInfo;

        uint256 gAccu = gr.totalStakedItem == 0 ?
                gr.accuedReward :
                gr.accuedReward + (block.number - gr.accuedToBlock) *
gr.rewardPerBlock / gr.totalStakedItem;
        ...
    }
```

contracts/smspool/CouplePool.sol#L421,L444

```
function getSocialPendingReward(address staker, uint256 matchCode) public view
override returns(uint256) {
        ...
        uint256 gAccu = gr.totalStakedItem == 0 ?
        ...
    }

    function getCustodialPendingReward(address staker, uint256 matchCode) public view
override returns(uint256) {
        ...
        uint256 gAccu = gr.totalStakedItem == 0 ?
        ...
    }
```

contracts/smspool/GroupPool.sol#L325

```
function getSocialPendingReward(address staker, uint256 matchCode) public view
override returns(uint256) {
        ...
        uint256 gAccu = gr.totalStakedItem == 0 ?
        ...
    }
```

Solution

It is recommended to check when `selfStakeInfos[staker].amount` or `totalStakedItem` is 0 and directly return 0 rewards in both cases.

Status

Fixed

[N18] [Critical] Unexpected rewards when staking in pairs

Category: Design Logic Audit

Content

In the CouplePool contract, the stakingManager can stake a user's NFT through the socialStake function. When the staking has not been paired yet, CouplePool will custody the user's NFT to SinglePool to obtain SinglePool staking rewards. Once the pairing is complete, it will withdraw from SinglePool and stake in CouplePool. Theoretically, during

the process of pairing, users should only receive rewards from SinglePool and not from CouplePool. Unfortunately, when SinglePool custody is performed, the user's `ssInfo.stakeInfo.amount` value in the CouplePool contract will be updated to the staked amount. This allows users to claim CouplePool staking rewards through the claimSocialReward function of the stakingManager contract even before the pairing is completed. Worse still, the user's `ssInfo.stakeInfo.claimedToAccued` has not been set at this point, so when settling rewards, `calculateStakeReward` will distribute large unexpected rewards to the user. Malicious users can exploit this issue to exhaust all reward tokens.

Similarly, this issue also exists in the GroupPool contract. Users can still claim large rewards from GroupPool even before the three-party pairing is completed.

Code location:

contracts/smspool/CouplePool.sol#L86

contracts/smspool/CouplePool.sol#L299

contracts/smspool/Pool.sol#L101

contracts/smspool/GroupPool.sol#L65

```
function socialStake(uint256 matchCode, address staker, uint8 nftId, uint256 amount)
        external
        override
        onlyManager
        returns(CustodialReward[2] memory custodialRewards)
    {
        SocialStakeInfo2 storage ssInfo = socialStakeInfos[matchCode];
        if (ssInfo.nftId1 == NftID.InvalidToken) {
            ssInfo.staker1 = staker;
            ssInfo.nftId1  = nftId;
            ssInfo.stakeInfo.amount = amount;
            ...
        }
    }

    function updateCustodialRewardInfo(uint256 matchCode) internal {
        CustodialInfo2 memory cusInfo = custodialInfos[matchCode];

        uint256 pendingReward = calculateStakeReward(cusInfo.stakeInfo);
        ...
    }

    function calculateStakeReward(StakeInfo memory stakeInfo) internal view
```

```
returns(uint256) {
        if (stakeInfo.amount == 0) return 0;

        uint256 deltaAccuReward = gRewardInfo.accuedReward -
stakeInfo.claimedToAccued;
        uint256 rewardAmount    = deltaAccuReward * stakeInfo.amount;

        return rewardAmount;
    }
```

Solution

It is recommended not to set the user's `ssInfo.stakeInfo.amount` value when the pairing has not been completed.

Status

Fixed

[N19] [Medium] Manipulate boost to influence the token id obtained in the auction

Category: Design Logic Audit

Content

In the Auction contract, when a user claims the auctioned NFT, the token id of the NFT depends not only on the random number seed but also on the amount of the user's deposit in the RFGDeposit contract. The larger the user's deposit amount, the greater the user's boost, and the higher the probability of obtaining a high-value NFT. Unfortunately, the calculation of the boost only depends on the user's deposit amount. Users can increase their RFG deposit before claiming the NFT to improve the probability. When multiple addresses of a user have obtained NFTs, they only need to withdraw the staked RFG tokens from other addresses and transfer them to the address that needs to claim the NFT for staking before claiming the NFT, in order to increase the probability. In other words, users only need a high amount of staking and can continuously stake/unstake/transfer RFG tokens to increase the probability of obtaining high-value NFTs at a lower cost.

Code location:

contracts/auction/Auction.sol#L290

```
function fakeRandomToken(address to, uint256 seed) internal view returns(uint8) {
        // range of boost is [1E18, 2E18]
        uint256 boost = IRFGDeposit(rfgDepositAddress).getDepositBoost(to);
```

```
        uint256 adjustBoost = (boost - 1E18) * 10; // enlarge 10 times of raw boost
        uint256 enlarge = adjustBoost * 30 / 1E18;
        enlarge = enlarge > 30 ? 30 : enlarge;
        uint256 tokenARange = 50 - enlarge;
        uint256 tokenBRange = 80 - enlarge;
        ...
    }
```

contracts/deposites/RFGDeposit.sol#L122

```
function calculateBoost(
        uint256 amount,
        uint256 lowerbound,
        uint256 upperbound,
        uint256 factor
    ) internal pure returns(uint256) {
        uint256 boost;

        if (amount <= lowerbound) {
            boost = 1E18;
        } else if (amount >= upperbound) {
            boost = 2E18;
        } else {
            boost = 1E18 + (amount - lowerbound) * factor / (upperbound - lowerbound);
            boost = boost > 2E18 ? 2E18 : boost;
        }

        return boost;
    }
```

Solution

It is recommended that when calculating the boost, the user's staking time should be taken into consideration. A lower-cost solution is to snapshot the boost of each eligible user off-chain and use it as part of the Merkle proof, participating in the token id calculation with a fixed value instead of obtaining it in real-time. However, this undoubtedly increases the risk of excessive privileges for the project team.

Status

Acknowledged; After communicating with the project team, the project team stated that they allow users to perform this operation.

[N20] [Suggestion] Redundant PoolMax enum

Category: Others

Content

In the NftID library, PoolID lists an enumeration of all the pools supported by the protocol, but PoolMax is not used anywhere in the protocol, which is redundant.

Code location: contracts/smspool/Commons.sol#L22

```
enum PoolID {
    ...
    PoolMax
}
```

Solution

It is recommended to remove the redundant PoolMax enumeration.

Status

Fixed

[N21] [Information] Reward calculation for two stakers in GroupPool being the same user

Category: Design Logic Audit

Content

In the GroupPool contract, when a user performs a socialUnstake/forceSocialUnstake operation, a portion of the bailed rewards of the initiator of the unstaking operation will be deducted and distributed to other users in the same group. However, it should be noted that one of the users in the same group may also be the initiator because the protocol allows the same user to provide two different NFTs for GroupPool staking. This means that a portion of the initiator's penalized rewards still belong to the initiator themselves.

Code location:

contracts/smspool/GroupPool.sol#L130

contracts/smspool/GroupPool.sol#L181

```
    function socialUnstake/forceSocialUnstake(...) external override onlyManager
returns(UnstakeReward[] memory) {
        ...
        if (initiator == ssInfo.staker1) {
```

```
            ...
        } else if (initiator == ssInfo.staker2) {
            ...
        } else {
            ...
        }
        ...
    }
```

contracts/smspool/StakingManager.sol#L192-L196

```
    function createMatch3Code(
        PoolID           pid,
        MatchPair memory p1,
        MatchPair memory p2,
        MatchPair memory p3,
        uint256          amount
    ) external {
        ...
        // not allowed: p1.staker == p2.staker == p3.staker
        require(
            p1.staker != p2.staker || p2.staker != p3.staker,
            "can not match self"
        );
        ...
    }
```

Solution

If this is not an intended design, it is recommended to check whether the initiator user has provided two NFTs when unstaking and deduct their respective Bailed rewards accordingly.

Status

Acknowledged

[N22] [Low] Potentially incorrect social staking reward information in GroupPool

Category: Design Logic Audit

Content

As previously mentioned, GroupPool allows the same user to provide two NFTs for staking. However, during reward settlement, stakerShareReward and bailed are calculated based on three different staking users. Therefore, in the viewSocialStakeRewardInfo function, when obtaining the user's pendingRewards, it only considers the scenario

where the three stakers are different users, while overlooking the possibility that two of the stakers might be the same user. This may cause the reward amount returned by the viewSocialStakeRewardInfo function to be lower than expected.

Code location: contracts/smspool/GroupPool.sol#L294-L296

```
    function viewSocialStakeRewardInfo(uint256 matchCode, address staker, uint256
gAccu) internal view returns(uint256) {
        ...
        if (staker == ssInfo.staker1)      return pendingRewards[matchCode][staker] +
p.staker1ShareReward - p.bailed1;
        else if (staker == ssInfo.staker2) return pendingRewards[matchCode][staker] +
p.staker2ShareReward - p.bailed2;
        else                               return pendingRewards[matchCode][staker] +
p.staker3ShareReward - p.bailed3;
    }
```

Solution

It is recommended to handle the case where two of the three stakers are the same user.

Status

Fixed

[N23] [Low] Potential Denial of Service Risk

Category: Denial of Service Vulnerability

Content

In Pool, users can freely choose different pools for staking. Theoretically, users can stake their owned NFTs in pools of different types or in different matchCodes within the same pool. The pool uses OpenZeppelin's EnumerableSet library to record the pools or matchCodes that users have joined, and retrieves all the pools or matchCodes joined by users through the values interface of EnumerableSet when claiming rewards. It is important to note that the values operation copies the entire storage space to memory. If the user participates in a large number of pools or matchCodes, the values operation will generate significant gas costs, potentially exceeding the block's gasLimit and ultimately leading to DoS risks. Despite this, if a DoS issue arises, users can still avoid their rewards being locked by claiming rewards individually.

Code location:

contracts/smspool/SinglePool.sol#L104

```
    function claimAllCustodialReward(address staker) public override onlyManager
returns(uint256) {
        uint256 allReward;

        uint256[] memory matchCodes = custodialMatchCodes[staker].values();
        ...
    }
```

contracts/smspool/CouplePool.sol#L322

contracts/smspool/CouplePool.sol#L346

```
    function claimAllCustodialReward(address staker) public override onlyManager
returns(uint256) {
        uint256 allReward;

        uint256[] memory custodialMatches = custodialMatchCodes[staker].values();
        ...
    }
    function claimAllSocialReward(address staker) public override onlyManager
returns(uint256) {
        uint256 allReward;

        uint256[] memory joinedMatches = socialMatchCodes[staker].values();
        ...
    }
```

contracts/smspool/GroupPool.sol#L248

```
    function claimAllSocialReward(address staker) public override onlyManager
returns(uint256) {
        uint256 allReward;
        uint256[] memory joinedMatches = socialMatchCodes[staker].values();
        ...
    }
```

Solution

One feasible approach is to limit the number of pools or matchCodes that users can participate in for staking.

Alternatively, storage with lower gas costs, such as lists, can be utilized.

Status

Acknowledged

[N24] [Suggestion] The validity of the pid was not checked when creating the pool

Category: Design Logic Audit

Content

In the StakingManager contract, operators can create pools using the createPool function, but the validity of the passed-in pid value is not checked. Theoretically, the pid of a pool should only be between 1 and 7.

Code location: contracts/smspool/StakingManager.sol#L100

```
    function createPool(PoolID pid, address poolProxy, bytes memory data) external
onlyRole(OperatorRole) {
        require(pools[pid] == address(0), "pool have created");
        require(poolProxy != address(0), "invalid proxy address");
        ...
    }
```

Solution

It is recommended to check the validity of the pid when creating a pool to avoid creating pools with unexpected pid values.

Status

Fixed

[N25] [Suggestion] When creating a matchCode, it does not check whether the pool has been created.

Category: Design Logic Audit

Content

In the StakingManager contract, users can create matchCodes for social staking using the createMatch2Code and createMatch3Code functions. However, when creating a matchCode, there is no check to verify if the pool corresponding to the pid has already been created. If the pool has not been created, users will be unable to successfully create a matchCode, and no error message will be thrown, which may cause confusion for users.

Code location: contracts/smspool/StakingManager.sol#L138,L171

```
function createMatch2Code(
    PoolID           pid,
    MatchPair memory p1,
    MatchPair memory p2,
    uint256          amount
) external {
    ...
}

function createMatch3Code(
    PoolID           pid,
    MatchPair memory p1,
    MatchPair memory p2,
    MatchPair memory p3,
    uint256          amount
) external {
    ...
}
```

Solution

It is recommended to check if the pool corresponding to the pid has already been created within the createMatch2Code and createMatch3Code functions. If the pool has not been created, detailed error messages should be thrown.

Status

Fixed

[N26] [Suggestion] There is an upper limit on the matchCodes available in the pool

Category: Design Logic Audit

Content

In the StakingManager contract, when a user creates a matchCode for social staking, the protocol assigns a matchCode to this staking. The matchCode is obtained through the nextMatchCode function of the pool, which is calculated using `poolID * 10 ** 8 + matchCodeNonce`. It is important to note that if the value of matchCodeNonce exceeds 1e8, it will affect the matchCode of the next pool. In reality, it is highly unlikely for a pool to have 1e8 matchCodes, but the project team should still remain attentive to this matter.

Code location: contracts/smspool/Pool.sol#L44

```
    function nextMatchCode() external override onlyManager returns(uint256) {
        ++matchCodeNonce;
        return uint256(poolID) * 10 ** 8 + matchCodeNonce;
    }
```

Solution

Alternative and more appropriate methods to calculate the matchCode may be worth considering.

Status

Fixed; The project team has increased the MatchCode limit to 1e10.

[N27] [Suggestion] Checks-Effects-Interactions are not followed when transferring out NFT

Category: Design Logic Audit

Content

In the StakingManager contract, the returnNftsBackAndClaimReward function is used to transfer users' staked NFTs from the contract back to the users and claim social staking rewards for users through the claimSocialReward function. The practice of transferring assets before modifying the contract state does not comply with the Checks-Effects-Interactions pattern. Although it does not lead to reentrancy risks in the current business scenario, it cannot be guaranteed that new exploitable business scenarios will not be introduced in the future.

Code location: contracts/smspool/StakingManager.sol#L435-L447

```
    function returnNftsBackAndClaimReward(uint256 matchCode) internal
returns(UnstakeBenefit[] memory benefits) {
        ...
        for (uint i; i < len;) {
            uint8[2] memory nftIds = mcode.findIdByAddress(joinedStakers[i]);
            transferOutNft(mcode.poolID, nftIds[0], joinedStakers[i], amount);

            if (nftIds[1] != NftID.InvalidToken) {
                transferOutNft(mcode.poolID, nftIds[1], joinedStakers[i], amount);
            }

            uint256 benefit = claimSocialReward(mcode.poolID, matchCode,
joinedStakers[i]);
            benefits[i] = UnstakeBenefit(joinedStakers[i], benefit);

            unchecked { ++i; }
```

```
        }
    }
```

Solution

It is recommended to use two for loops: first, use the claimSocialReward function to settle rewards, and then use another for loop to transfer the NFTs.

Status

Confirmed

[N28] [Suggestion] Optimizable reward information update

Category: Gas Optimization Audit

Content

In the updateRewardInfo function of the GroupPool contract, the currently claimable social staking rewards are calculated through the calculateStakeReward function, and the rewards are distributed to the stakers. It is important to note that when users exit staking through the StakingManager contract, multiple calls to the updateRewardInfo function may be involved in a single transaction. The pendingReward for reward settlement is only greater than 0 during the first call, and when pendingReward is 0, the updateRewardInfo function still performs reward distribution operations, which will consume a lot of unnecessary gas.

Code location: contracts/smspool/GroupPool.sol#L216

```
    function updateRewardInfo(uint256 matchCode) internal {
        SocialStakeInfo3 memory ssInfo = socialStakeInfos[matchCode];

        uint256 pendingReward = calculateStakeReward(ssInfo.stakeInfo);
        ...
    }
```

Solution

It is recommended to check if pendingReward is greater than 0 in the updateRewardInfo function and only distribute rewards when pendingReward is greater than 0.

Status

Fixed

[N29] [Medium] Risks of excessive privilege

Category: Authority Control Vulnerability Audit

Content

In the StakingManager contract, the admin role can upgrade any pool through the upgradePool function. Moreover, in the protocol, except for the InviteReward, Airdrop, RFGToken, and NftCard contracts, all other contracts use an upgradable model, where the admin of the proxy contract can arbitrarily upgrade these contracts. This leads to the risk of excessive privileges.

In the Auction contract, after the auction is completed, the project team will calculate off-chain the users who can obtain NFTs and the final auction price, and establish a Merkle proof for users to claim. This also increases the centralization risk to a certain extent.

Code location:

contracts/smspool/StakingManager.sol#L95

```
    function upgradePool(PoolID pid, address newImpl, bytes memory data) external
onlyRole(AdminRole) {
        require(1 <= uint8(pid) && uint8(pid) <= 7, "invalid pool id");

        IPoolUpgradable(pools[pid]).upgradeToAndCall(newImpl, data);

        emit PoolUpgraded(pid, newImpl, data);
    }
```

contracts/auction/Auction.sol#L180-L183

```
    function finishAuction(uint8 aucId, bytes32 merkleRoot, uint256 totalSold,
uint256 price) external onlyRole(OperatorRole) {
        ...
        AuctionResult storage result = auctionResults[aucId];
        result.merkleRoot = merkleRoot;
        result.totalSold  = totalSold;
        result.price      = price;
        ...
    }
```

Solution

From a short-term perspective, to ensure the stable operation of the project in its early stages, assigning the above

privileged roles to a multi-signature wallet can effectively address the single point of failure risk, but it still cannot mitigate the risk of excessive privileges. In the long run, when the project is running stably, transferring the protocol's privileged roles to community governance can effectively alleviate the risk of excessive privileges.

Status

Acknowledged; After communicating with the project team, they stated that once the protocol is deployed, the project team will use multisig for privilege management to mitigate single-point-of-failure risk. In the future, after the protocol has been running stably, community governance will be enabled to thoroughly resolve the risk of excessive centralization.

[N30] [Suggestion] Protocol Missing Emergency Operations Role

Category: Authority Control Vulnerability Audit

Content

The protocol has planned for multiple roles to manage different contracts, but it is important to note that the protocol lacks an emergency pause functionality and a role to manage this function. When an emergency occurs in the protocol, the emergency operation role can close the protocol through the pause function to minimize losses as much as possible.

Solution

It is recommended that the protocol add an emergency pause functionality and a role to manage this function. The emergency operation role can be assumed by an EOA to quickly handle emergency situations without the need to contact other team members. It should also be noted that this role should only be used to manage the pause permissions of the contracts and should not have any overlap with other permissions.

Status

Fixed

[N31] [Suggestion] Missing event records

Category: Others

Content

In the NftCard contract, the owner can modify the URI of the NFT through the setUri function, but no event is recorded.

Code location: contracts/tokens/NftCard.sol#L80

```
function setUri(string memory newUri) external onlyOwner {
    _setURI(newUri);
}
```

Solution

It is recommended to record events for modifications of sensitive parameters to facilitate future self-inspection or community auditing.

Status

Fixed

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002408130003	SlowMist Security Team	2024.07.29 - 2024.08.13	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 4 critical risks, 1 high risk, 4 medium risks, 6 low risk, 13 suggestions, and 3 information. 1 suggestion were confirmed; All other findings were fixed or acknowledged. The code was not deployed to the mainnet. Since the protocol has not been deployed yet, its excessive privilege issue remains unresolved, and therefore, the protocol is still at medium risk. Once the protocol is deployed, the project team will use multisign wallet for privilege management to mitigate single-point-of-failure risk. After the protocol has been running stably, community governance will be enabled to thoroughly resolve the risk of excessive centralization.

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