Smart Contract Security Audit Report

Contents

1. Executive Summary

On Dec. 22, 2020, the SlowMist security team received the SIL Finance team's security audit application for Sister in Law, 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.

SlowMist Smart Contract DeFi project test method:

Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code module 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.

SlowMist Smart Contract DeFi project risk level:

Critical vulnerabilities	Critical vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High-risk vulnerabilities	High-risk vulnerabilities will affect the normal operation of DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium-risk vulnerabilities	Medium vulnerability will affect the operation of DeFi project. It is recommended to fix medium-risk vulnerabilities.

Low-risk vulnerabilities	Low-risk vulnerabilities may affect the operation of DeFi project in certain scenarios. It is suggested that the project party should evaluate and consider whether these vulnerabilities need to be fixed.
Weaknesses	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Enhancement Suggestions	There are better practices for coding or architecture.

2. Audit Methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and in-house 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:

- Reentrancy attack and other Race Conditions
- Replay attack
- Reordering attack
- Short address attack
- Denial of service attack
- Transaction Ordering Dependence attack
- Conditional Completion attack
- Authority Control attack
- Integer Overflow and Underflow attack

- TimeStamp Dependence attack
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Explicit visibility of functions state variables
- Logic Flaws
- Uninitialized Storage Pointers
- Floating Points and Numerical Precision
- tx.origin Authentication
- "False top-up" Vulnerability
- Scoping and Declarations

3. Project Background

3.1 Project Introduction

"Sister In Law" token a.k.a "SIL" is a decentralized automatic investment platform based on smart contracts, focusing on providing users with DeFi Financial Management services. SIL provides dual-token liquidity for variable swaps, automatic LP matching, and automatic compound interests. According to factors such as annualized rate of return, safety factor, financial management cycle, etc., it automatically selects and configures products that best suit the interests of users, make complex liquidity mining to become simple. The revenue of mining will be distributed to all users in proportion, there is no intermediary, no principal commission fees. It's fair and just. The platform is jointly built by crypto enthusiasts from all over the world, and the management of the platform is entrusted to all SIL holders.

Audit version file information

Initial audit files:

sil_contract.zip(SHA256):

c50905a92ecc8b047a69ac08a86636ba5374250aaa8bac80fcce7da241d1905e

Final audit files:

contract.zip(SHA256):

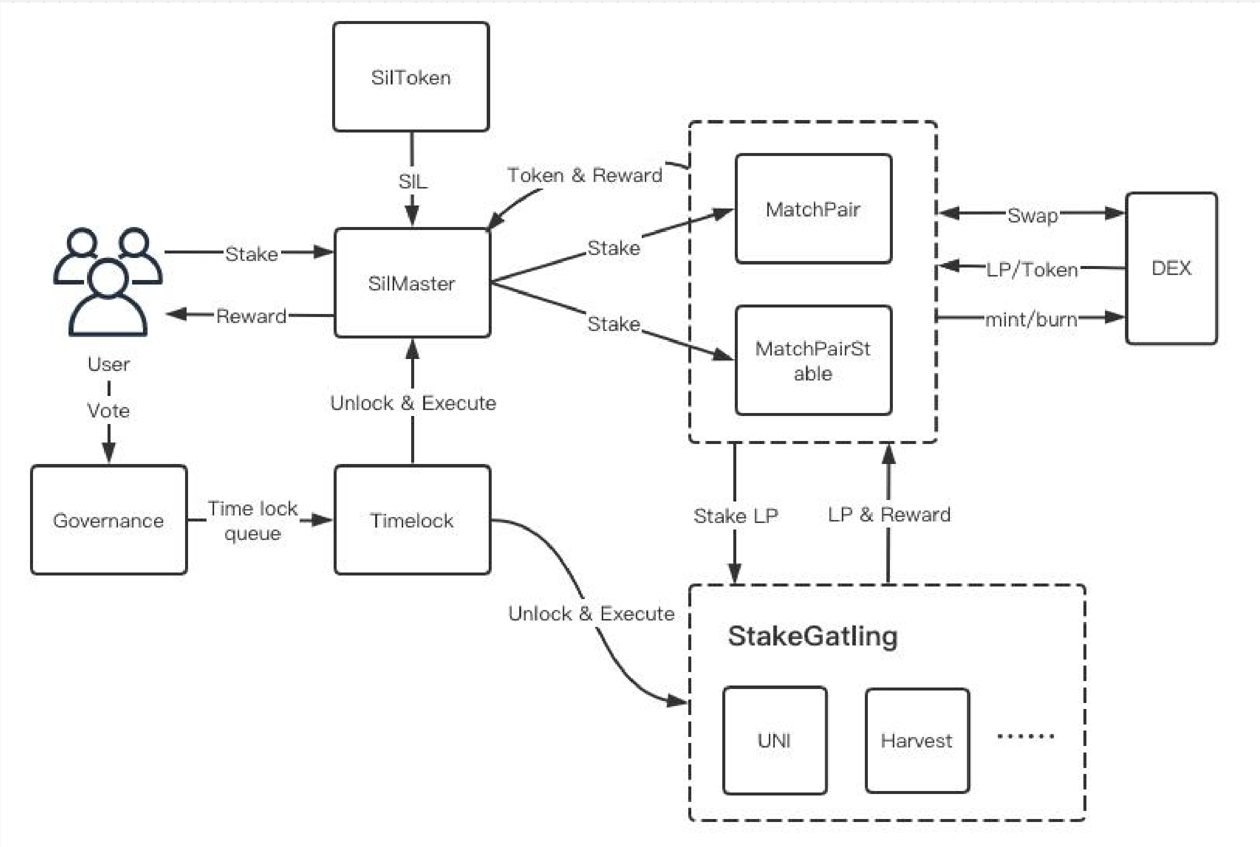
356ab2e95527749684d6a92fe3165536431ce1bac5ea66acb13034a67787a65d

3.2 Project Structure

```
├── MatchPair.sol
├── MatchPairStable.sol
├── PriceChainLinkCheker.sol
├── PriceSafeCheker.sol
├── SilMaster.sol
├── SilToken.sol
├── StakeGatling.sol
├── StakeGatlingHarvest.sol
├── TrustList.sol
├── chainlink
│   └── APIConsumerToUint.sol
├── interfaces
│   ├── IMatchPair.sol
│   ├── IMigrateGatling.sol
│   ├── IMintRegulator.sol
│   ├── IPriceSafeChecker.sol
│   ├── IStakeGatling.sol
│   ├── IStakingRewards.sol
│   ├── IUniswapV2Router01.sol
│   └── IWETH.sol
└── utils
    ├── MasterCaller.sol
    ├── QueueStableStakesFuns.sol
    └── QueueStakesFuns.sol
```

3.3 Contract Structure

In the SIL Finance project, users can select the required impermanence model stake tokens through the SilMaster contract entry. The MatchPair contract will match the tokens of different users' stakes according to the impermanence model selected by the user, and provide liquidity to the DEX after the matching is completed. Obtain LP Token and pledge the obtained LP Token to StakeGatling. StakeGatling will use UNI, Harvest and other projects to mortgage the LP Token in the pool and obtain income, and then transfer the income to DEX for token exchange, and perform token matching to add liquidity to obtain LP Token, and then mortgage it into StakeGatling compound interest. In the end, the user will receive SIL rewards and the income obtained after successfully matching the LP and the income obtained by compound interest in StakeGatling when withdrawing, but the user will still bear a part of the risk of impermanent loss caused by adding liquidity. The specific architecture diagram of the project is as follows:



4. Code Overview

4.1 Main Contract address

The contract has not yet been deployed on the mainnet.

4.2 Contracts Description

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

MatchPair			
Function Name	Visibility	Mutability	Modifiers
setStakeGatling	Public	Can modify state	onlyOwner
stake	Public	Can modify state	onlyMasterCaller
updatePool	Private	Can modify state	-
getPairAmount	Public	Can modify state	-
untakeToken	Public	Can modify state	onlyMasterCaller
untakeLP	Private	Can modify state	-
queueTokenAmount	Public	-	-
balanceOfToken0	External	-	-
balanceOfToken1	External	-	-
balanceOfLP0	External	-	-
balanceOfLP1	External	-	-
token	Public	-	-
token0	Public	-	-
token1	Public	-	-
createQueue	Private	Can modify state	-
toQueue	Private	Can modify state	-
getQueuePoolInfo	Private	Can modify state	-
untakePending	Private	Can modify state	-
untakePriority	Private	Can modify state	-
pending2LP	Private	Can modify state	-

lp2PriorityQueueSpecifical	Private	Can modify state	-
lp2PriorityQueue	Private	Can modify state	-
moveQueue2LP	Private	Can modify state	-
appendLP	Private	Can modify state	-
appendPriority	Private	Can modify state	-
lPAmount	Public	-	-
tokenAmount	Public	-	-
lp2TokenAmount	Public	-	-
maxAcceptAmount	Public	-	-
getReserves	Public	-	-
setPriceSafeChecker	Public	Can modify state	onlyOwner

MatchPairStable			
Function Name	Visibility	Mutability	Modifiers
createQueue	Private	Can modify state	-
setStakeGatling	Public	Can modify state	onlyOwner
stake	Public	Can modify state	onlyMasterCaller
updatePool	Private	Can modify state	-
getPairAmount	Public	Can modify state	-
untakeToken	Public	Can modify state	onlyMasterCaller
distributePairedOrigin	Private	Can modify state	-
coverageLose	Private	Can modify state	-
execSwap	Private	Can modify state	-
toQueue	Private	Can modify state	-
getQueuePoolInfo	Private	Can modify state	-
untakePending	Private	Can modify state	-
untakePriority	Private	Can modify state	-
pending2LP	Private	Can modify state	-
burnLp	Private	Can modify state	-
lpOriginAccountCalc	Private	Can modify state	-
burnUserLp	Private	Can modify state	-
untakePairedLP	Private	Can modify state	-
moveQueue2LP	Private	Can modify state	-
getAmountVinIndexed	Private	Can modify state	-
getAmountVoutIndexed	Private	Can modify state	-
getAmountIn	Internal	-	-

getAmountOut	Internal	-	-
lPAmount	Public	-	-
tokenAmount	Public	-	-
lp2TokenAmount	Public	-	-
maxAcceptAmount	Public	-	-
queueTokenAmount	Public	-	-
balanceOfToken0	External	-	-
balanceOfToken1	External	-	-
balanceOfLP0	External	-	-
balanceOfLP1	External	-	-
token	Public	-	-
token0	Public	-	-
token1	Public	-	-
getReserves	Public	-	-
setPriceSafeChecker	Public	Can modify state	onlyOwner
setMinLimit	Public	Can modify state	onlyOwner
mergeArray	Private	-	-

SilMaster			
Function Name	Visibility	Mutability	Modifiers
poolLength	External	-	-
setMintRegulator	Public	Can modify state	onlyOwner
setMintRegulator	Public	Can modify state	onlyOwner
updateSilPerBlock	Public	Can modify state	-
add	Public	Can modify state	onlyOwner
holdWhaleSpear	Public	Can modify state	onlyOwner
set	Public	Can modify state	onlyOwner
getMultiplier	Public	-	-
pendingSil	External	-	-
massUpdatePools	Public	Can modify state	-
updatePool	Public	Can modify state	-
depositEth	Public	payable	-
deposit	Public	Can modify state	-
withdrawToken	Public	Can modify state	-
withdraw	Private	Can modify state	-
withdrawSil	Public	Can modify state	-

withdrawSilCalcu	Private	Can modify state	-
safeSilTransfer	Internal	Can modify state	-
mintableAmount	External	-	-
dev	Public	Can modify state	-
safeTransferFrom	Internal	Can modify state	-
safeTransfer	Internal	Can modify state	-
safeTransferETH	Internal	Can modify state	-
fallback	External	payable	-
getFeeRewardAmount	Private	-	-
notifyRewardAmount	External	Can modify state	onlyOwner

SilToken			
Function Name	Visibility	Mutability	Modifiers
mint	Public	Can modify state	onlyOwner
delegates	External	-	-
delegate	External	Can modify state	-
delegateBySig	External	Can modify state	-
getCurrentVotes	External	-	-
getPriorVotes	External	-	-
_delegate	Internal	Can modify state	-
_moveDelegates	Internal	Can modify state	-
_writeCheckpoint	Internal	Can modify state	-
safe32	Internal	-	-
getChainId	Internal	-	-
_transfer	Internal	Can modify state	-

StakeGatling			
Function Name	Visibility	Mutability	Modifiers
initApprove	Public	Can modify state	-
setMatchPair	Public	Can modify state	onlyOwner
setRouterPaths	Public	Can modify state	onlyOwner
stake	External	Can modify state	onlyMasterCaller
withdraw	Public	Can modify state	onlyMasterCaller
updateRate	Private	Can modify state	-

currentProfitRate	Public	-	-
presentRate	Public	-	-
reprofitCountAverage	Public	-	-
sellEarn2TokenTwice	Private	Can modify state	-
earnMinedToken	Private	Can modify state	-
mintLP	Private	Can modify state	-
getPairAmount	Public	-	-
profitRateDenominator	Public	-	-
getAmountOIn	Private	Can modify state	-
totalLp	Public	-	-
totalToken	Public	-	-

TrustList			
Function Name	Visibility	Mutability	Modifiers
updateList	Public	Can modify state	onlyMasterCaller
trustable	Internal	Can modify state	onlyOwner

PriceSafeCheker			
Function Name	Visibility	Mutability	Modifiers
checkPrice	External	-	-
getLatestPrice	Public	-	-
feedPrice	Public	Can modify state	onlyOwner
setPriceRange	Public	Can modify state	onlyOwner

PriceChainLinkCheker			
Function Name	Visibility	Mutability	Modifiers
getLatestPrice	Public	-	-
checkPrice	External	-	-
setPriceRange	Public	Can modify state	onlyOwner

4.3 Code Audit

4.3.1 Critical vulnerabilities

4.3.1.1 Risk of Token Mismatch

The user can pledge ETH or USDT through the deposit entry of the SilMaster contract, and then the contract will call the stake function of the MatchPair contract to perform the stake operation. First, the pledged tokens will be added to the queue, and then the updatePool function will be called to update the pool. The updatePool function will first obtain the required number of matches through the getPairAmount interface, then put the matched tokens into Uniswap and call the mint function to add liquidity.

But this will lead to: For example, there are 100 ETH in the queue waiting to be matched, and 0 USDT. At this time, the attacker can first go to a certain ETH pool of uniswap (such as WETH-WBTC) to borrow WETH by the flash loan, and perform the swap operation (WETH->USDT) with the borrowed WETH in the WETH-USDT pool of uniswap. At this time, the slippage of this pool will increase. Next, the attacker uses USDT to perform mortgage matching on SILFinance. Since the number of ETH matched by USDT is calculated by calling the getReserves interface of the pair contract, this interface takes the real-time tokens in the pair pool. The number of tokens in the pool has been out of balance after the previous Swap operation, so the number of ETH calculated here will inevitably be more than normal. Therefore, the attacker can use the same USDT matching queue with more ETH to add liquidity. Finally, the attacker can successfully arbitrage by performing the Swap (USDT->WETH) operation in reverse.

Fix suggestion: You can use the getReserves price feed interface implemented by yourself, and you can refuse the operation if the deviation is too large to compare with the contract acquisition by obtaining credible data.

Code location: MatchPair.sol & MatchPairStable.sol

```
function getPairAmount(
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired   ) public returns ( uint amountA, uint amountB) {

    (uint reserveA, uint reserveB,) = lpToken.getReserves(); //SlowMist// Obtain the real-time quantity of the two tokens of the Pair contract, but did not check.

    uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA, reserveB);
    if (amountBOptimal <= amountBDesired) {
        (amountA, amountB) = (amountADesired, amountBOptimal);
    } else {
        uint amountAOptimal = UniswapV2Library.quote(amountBDesired, reserveB, reserveA);
        assert(amountAOptimal <= amountADesired);
        (amountA, amountB) = (amountAOptimal, amountBDesired);
    }
}
```

Fix status: Fixed.

4.3.2 High-risk vulnerabilities

4.3.2.1 Risk of Delegation Double Spending

There has a delegation double spending bug, The governance contract allows token holders to give voting power to a delegate. but there has a bug, voting power stays with the delegate even when the token holder transfers the tokens from the wallet.

Fix suggestion: In this case, the voting delegation should disappear. Instead, the voting power can be inflated with delegate & transfer transactions.

Code location: SilToken.sol

```
    function _transfer(address sender, address recipient, uint256 amount) internal virtual {
    //SlowMist// When the token was transferred, the corresponding voting rights were not transferred.
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _beforeTokenTransfer(sender, recipient, amount);

        _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }
```

Fix status: Fixed

4.3.3 Medium-risk vulnerabilities

4.3.3.1 Risk of Contract Denial of Service

When calling the massUpdatePools function in the SilMaster contract to update all pools, it will update all pools through a for loop. If the number of pools exceeds its recursion depth, the contract call will fail.

Fix suggestion: It is suggested to use mapping instead of for loop.

Code location: SilMaster.sol

```
function massUpdatePools() public {
    uint256 length = poolInfo.length;
    for (uint256 pid = 0; pid < length; ++pid) {
        updatePool(pid);
    }
}
```

Fix status: After communicating with the project party, the project party will not add too many pools in the future, and the project party will avoid this risk by controlling the number of pools.

4.3.3.2 Risk of Excessive Authority

In the SilMaster contract, the owner role can add pools, modify pool weights, modify sensitive parameters, etc., which will lead to the risk of excessive owner authority. The project party reported that this part of the authority will be transferred to the timelock contract, and the timelcok contract authority will be transferred to the community governance.

In the SilToken contract, the owner role can mint arbitrarily through the mint function, which will lead to the risk of excessive owner authority. The project party reported that this part of the authority will be transferred to the SilMaster contract.

In the StakeGatling contract, the owner role can arbitrarily modify the matchPair contract address and RouterPath, which will lead to the risk of excessive owner authority. It is recommended to transfer owner permissions to the timelock contract.

Fix suggestion: It is suggested to transfer sensitive operation permissions involving user assets to the community for governance.

Fix status: After the project party communicated and feedback, the project party stated that the sensitive operation authority related to user assets will be transferred to the timelcok contract, and the timelock contract will be governed by the community's voting.

4.3.4 Low-risk vulnerabilities

4.3.4.1 WhaleSpear function design flaw

The SilMaster contract has the whaleSpear function. When whaleSpear is turned on, the contract will check whether the number of tokens deposited by the user exceeds a maximum allowable deposit value (maxAcceptAmount), and the contract incorrectly subtracts the strategy pool when calculating the maximum allowable deposit value The number of tokens corresponding to the mortgaged LP, without subtracting the number of tokens remaining in the pending queue.

Fix suggestion: When calculating the maximum deposit value, the number of tokens in the pending queue should be subtracted.

Code location: SilMaster.sol

```
    function maxAcceptAmount(uint256 _index, uint256 _times, uint256 _inputAmount) public view override returns
(uint256) {

        QueuePoolInfo storage info =   _index == 0? queueStake0: queueStake1;
        (uint256 amount0, uint256 amount1) = stakeGatling.totalToken();

        uint256 pendingTokenAmount = info.totalPending;
        uint256 lpTokenAmount =   _index == 0 ? amount0 : amount1;

        uint256 maxAmount = lpTokenAmount.mul(_times).sub(lpTokenAmount);


        if(maxAmount > 0) {
            return _inputAmount > maxAmount ? maxAmount : _inputAmount ;
        }else {
           return _inputAmount;
        }
```

```
    }
```

Fix status: Fixed.

4.3.5 Enhancement Suggestions

4.3.5.1 Event missing

In the APIConsumerToUint contract, the requestSetting function is used to modify key parameters, such as _oracle, _jobId, _fee, etc., but no event record is made during the modification.

Fix suggestion: It is suggested to record events when modifying key parameters to facilitate subsequent operation audits.

Code location: APIConsumerToUint.sol

```
    function requestSetting(
        address _oracle,
        bytes32 _jobId,
        string calldata _url,
        string calldata _path,
        int256 _times,
        uint256 _fee)
        public onlyOwner()
        returns (bytes32 requestId)
    {
        oracle = _oracle;
        jobId   = _jobId;
        url     = _url;
        path    = _path;
        times   = _times;
        fee = _fee;
        // fee = 0.1 * 10 ** 18; // 0.1 LINK
        lastUpdateTime = 0;
```

```
        //SlowMist// No event record was made when sensitive parameter modification was
made
    }
```

Fix status: Fixed.

4.3.5.2 Enhancement Point of DelegateBySig Function

The nonce in the delegateBySig function is input by the user. When the user input a larger nonce, the current transaction cannot be success but the relevant signature data will still remain on the chain, causing this signature to be available for some time in the future.

Fix suggestion: It is suggested to fix it according to EIP-2612.

Code location: SilToken.sol

```
    function delegateBySig(
        address delegatee,
        uint nonce,
        uint expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
        external
    {
        bytes32 domainSeparator = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name())),
                getChainId(),
                address(this)
            )
        );

        bytes32 structHash = keccak256(
```

```
                abi.encode(
                    DELEGATION_TYPEHASH,
                    delegatee,
                    nonce,
                    expiry
                )
            );

            bytes32 digest = keccak256(
                abi.encodePacked(
                    "\x19\x01",
                    domainSeparator,
                    structHash
                )
            );

            address signatory = ecrecover(digest, v, r, s);
            require(signatory != address(0), "CYZ::delegateBySig: invalid signature");
            require(nonce == nonces[signatory]++, "CYZ::delegateBySig: invalid nonce");
            require(now <= expiry, "CYZ::delegateBySig: signature expired");
            return _delegate(signatory, delegatee);
        }
```

Fix status: After communicating with the project party, this issue does not affect the business logic and will not be fixed temporarily.

4.3.5.3 The change of LP pool weights affects users' income

In the SilMaster contract, when the Owner calls the add function and the set function to add a new pool or reset the pool weight, all LP pool weights will change accordingly. The Owner can update all pools before adjusting the weight by passing in the _withUpdate parameter with a value of true to ensure that the user's income before the pool weight is changed will not be affected by the adjustment of the pool weight, but if the value of the _withUpdate parameter is false, then All pools will not be updated before the pool weight is adjusted, which will cause the user's income to be affected before the pool weight is changed.

Fix suggestion: It is suggested to force all LP pools to be updated before the weights of LP pools are adjusted to avoid the impact of user income..

Code location: SilMaster.sol

```
function add(uint256 _allocPoint, IMatchPair _lpToken, bool _withUpdate) public onlyOwner {

        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            totalDeposit0: 0,
            totalDeposit1: 0,
            accSilPerShare0: 0,
            accSilPerShare1: 0
            }));
    }

    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Fix status: Fixed.

4.3.5.4 Risk of Potential Token Transfer Failure

When the user deposits the token, the safeTransferFrom function is used to transfer the corresponding token, and the safeTransfer function is used to transfer the token when withdrawToken. The safeTransferFrom function and safeTransfer function will check the returned success and data , If the connected token defines the return value, but does not return according to the EIP20 specification, the user will not be able to pass the check here, resulting in the tokens being unable to be transferred in or out.

Fix suggestion: It is suggested that when docking new tokens, the project party should check whether its writing complies with EIP20 specifications.

Code location: SilMaster.sol

```
     function safeTransferFrom(address token, address from, address to, uint value) internal {
         // bytes4(keccak256(bytes('transferFrom(address,address,uint256)')));
         (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x23b872dd, from, to, value));
         require(success && (data.length == 0 || abi.decode(data, (bool))), 'MasterTransfer: 
TRANSFER_FROM_FAILED');
    }

    function safeTransfer(address token, address to, uint value) internal {
         // bytes4(keccak256(bytes('transfer(address,uint256)')));
         (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0xa9059cbb, to, value));
         require(success && (data.length == 0 || abi.decode(data, (bool))), 'MasterTransfer: TRANSFER_FAILED');
    }
```

Fix status: After communicating with the project party, the project party will strictly check whether the writing of the new token complies with the EIP20 specification when docking the new token.

4.3.5.5 Failure to follow the Checks-Effects-Interactions principle

When users withdrew ETH, they did not strictly follow the principle of first verifying and then changing the status before invoking transfers. There is a reentrancy issue, but it cannot cause harm to the project.

Fix suggestion: It is suggested to implement in strict accordance with coding standards.

Code location: SilMaster.sol

```
function withdrawToken(uint256 _pid, uint256 _index, uint256 _amount) public override {
    address _user = msg.sender;
    PoolInfo storage pool = poolInfo[_pid];
    //withdrawToken from MatchPair
    uint256 untakeTokenAmount = pool.lpToken.untakeToken(_index, _user, _amount);
    address targetToken = pool.lpToken.token(_index);
    uint256 userAmount = untakeTokenAmount.mul(995).div(1000);
    if(targetToken == WETH) {
        IWETH(WETH).withdraw(untakeTokenAmount);
        safeTransferETH(_user, userAmount);
        safeTransferETH(repurchaseaddr, untakeTokenAmount.sub(userAmount) );
    }else {
        safeTransfer(pool.lpToken.token(_index),   _user, userAmount);
        safeTransfer(pool.lpToken.token(_index),   repurchaseaddr, untakeTokenAmount.sub(userAmount));
    }
    withdraw(_pid, _index, _user, untakeTokenAmount);
}
```

Fix status: Fixed.

4.3.5.6 Part of the code is redundant

In the StakeGatling contract, the reserve parameter retrieved by the getReserves interface of the sellUNI2TokenTwice function is not used in this function, as is the mintLP function. The balance0 and balance1 variables of the sellUNI2TokenTwice function in the StakeGatlingHarvest contract are also not used. The visibility of sellExrateToken2Token1, sellUNI2Token function is private, but no other contract is calling.

Fix suggestion: If the function and variable are not used, it is suggested to remove

Fix status: Fixed.

5. Audit Result

5.1 Conclusion

Audit Result : There is a risk of excessive authority

Audit Number : 0X002101050003

Audit Date : Jan. 05, 2021

Audit Team : SlowMist Security Team

Summary conclusion: The SlowMist security team use a manual and SlowMist Team analysis tool audit of the codes for security issues. There are eleven security issues found during the audit. There are one critical-risk vulnerabilities, one high-risk vulnerabilities, two medium-risk vulnerabilities and one low-risk vulnerabilities. We also provide six enhancement suggestions. As the SIL project has not yet been deployed on the mainnet, and the authority of each contract has not been transferred to the community governance, the project still has the risk of excessive authority.

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 base 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 this 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.

SLOWMIST

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

Github
https://github.com/slowmist