Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.09.12, the SlowMist security team received the GoPlus team's security audit application for SafeToken Protocol, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

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

The test method information:

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

The vulnerability severity level information:

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

2 Audit Methodology

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

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

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

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

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

3 Project Overview

3.1 Project Introduction

This audit primarily focuses on four smart contracts of the SafeToken protocol: SafeTokenFactory, TokenTemplate, TokenLocker, and UniV3LPLocker. Among these, SafeTokenFactory is used for creating tokens using templates, where it can be seen that the token requires setting up fees and whitelists/blacklists, which are utilized in the transfer function. TokenLocker and UniV3LPLocker are mainly used for token lock-up, allowing users to transfer ERC20 tokens or LP tokens into the contract for locking. Users can only unlock and withdraw their tokens after the lock-up period ends.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Lack of non-zero address check	Others	Suggestion	Fixed
N2	Risk of front-running attack	Authority Control Vulnerability Audit	Medium	Fixed
N3	Lack of check on the blacklist setting	Design Logic Audit	Medium	Fixed
N4	Missing the event records	Others	Suggestion	Fixed
N5	Missing scope limit	Others	Low	Fixed
N6	Token compatibility issues	Design Logic Audit	Low	Fixed
N7	lockInfo.amount is not set to zero	Design Logic Audit	Suggestion	Fixed
N8	Permission check repetition	Authority Control Vulnerability Audit	Suggestion	Fixed
N9	Repeat adding the names of supported fee information	Design Logic Audit	Suggestion	Fixed
N10	Signature replay risk	Replay Vulnerability	Medium	Fixed
N11	Missing chainID check in the signature verification	Replay Vulnerability	Medium	Fixed
N12	Potential Sandwich Attack Risk	Reordering Vulnerability	Low	Acknowledged
N13	Redundant code	Others	Suggestion	Fixed
N14	No handling fee will be settled first when transferring the lock permission	Design Logic Audit	Information	Acknowledged
N15	Blacklist and whitelist cannot be changed after setting	Design Logic Audit	Information	Acknowledged
N16	Risk of excessive authority	Authority Control Vulnerability Audit	Medium	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/GoPlusSecurity/SafeToken-Protocol

commit: ddcd395e0b0846a39dc4807489ec0d575130d43a

Fixed Version:

https://github.com/GoPlusSecurity/SafeToken-Protocol

commit: b417494224b21cc97b958942634fa7b9df28009a

Audit scope:

- contracts/SafeTokenFactory.sol
- contracts/TokenTemplate.sol
- contracts/TokenLocker.sol
- contracts/UniV3LPLocker.sol
- interface/*.sol
- libs/*.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:

SafeTokenFactory			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
cumputeTokenAddress	External	-	-

SafeTokenFactory			
createToken	External	Can Modify State	-
updateTemplates	External	Can Modify State	onlyOwner

TokenLocker			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
addOrUpdateFee	Public	Can Modify State	onlyOwner
updateFeeReceiver	External	Can Modify State	onlyOwner
_takeFee	Internal	Can Modify State	-
_addLock	Internal	Can Modify State	-
_createLock	Internal	Can Modify State	-
lock	External	Payable	nonReentrant
lockWithPermit	External	Payable	nonReentrant
_vestingLock	Internal	Can Modify State	-
vestingLock	External	Payable	nonReentrant
vestingLockWithPermit	External	Payable	nonReentrant
_updateLock	Internal	Can Modify State	-
updateLock	External	Payable	validLockOwner nonReentrant
updateLockWitPermit	External	Payable	validLockOwner nonReentrant
transferLock	External	Can Modify State	validLockOwner
acceptLock	External	Can Modify State	-
unlock	External	Can Modify State	validLockOwner nonReentrant
_normalUnlock	Internal	Can Modify State	-

TokenLocker			
_vestingUnlock	Internal	Can Modify State	-
_withdrawableTokens	Internal	-	-
withdrawableTokens	External	-	-
getUserNormalLocks	External	-	-
getUserLpLocks	External	-	-
getTokenLocks	External	-	-

SafeUniswapCall			
Function Name	Visibility	Mutability	Modifiers
checkIsPair	Public	-	-
safeCallFactory	Public	-	-
safeCallPair	Public	-	-
decodeRet2Address	Public	-	-

TokenTemplate			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	initializer
devInit	External	Can Modify State	-
_transfer	Internal	Can Modify State	-
burn	Public	Can Modify State	-
removeBlacklist	External	Can Modify State	onlyOwner
removeTax	External	Can Modify State	onlyOwner

UniV3LPLocker			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
addOrUpdateFee	Public	Can Modify State	onlyOwner
removeFee	External	Can Modify State	onlyOwner
updateFeeReceiver	External	Can Modify State	onlyOwner
updateFeeSigner	External	Can Modify State	onlyOwner
addSupportedNftManager	External	Can Modify State	onlyOwner
supportedNftManager	Public	-	-
isSupportedFeeName	Public	-	-
getFee	Public	-	-
_deductLockFee	Internal	Can Modify State	-
_getPool	Internal	-	-
lock	External	Payable	-
lockWithCustomFee	External	Payable	-
_verifySignature	Internal	-	-
_lock	Internal	Can Modify State	-
transferLock	External	Can Modify State	validLockOwner
acceptLock	External	Can Modify State	-
increaseLiquidity	External	Payable	nonReentrant
decreaseLiquidity	External	Payable	validLockOwner nonReentrant
unlock	External	Can Modify State	validLockOwner
relock	External	Can Modify State	validLockOwner nonReentrant

UniV3LPLocker			
_collect	Private	Can Modify State	-
collect	External	Can Modify State	nonReentrant
setCollectAddress	External	Can Modify State	validLockOwner nonReentrant
_getLiquidity	Private	-	-
adminRefundEth	External	Can Modify State	onlyOwner nonReentrant
adminRefundERC20	External	Can Modify State	onlyOwner nonReentrant
onERC721Received	External	-	-

4.3 Vulnerability Summary

[N1] [Suggestion] Lack of non-zero address check

Category: Others

Content

1.In the SafeTokenFactory contract, when the createToken function is called to create a new token, the parameters owner_ and dest_ that are passed in will be used to initialize the owner role address and the initial minting address of the token. However, there is a lack of non-zero address check here. If key address variables are mistakenly set to zero addresses, it could potentially lead to unexpected errors, such as token loss.

Code Location:

contracts/SafeTokenFactory.sol#L45

```
function createToken(
    uint256 tempKey_,
    string memory symbol_,
    string memory name_,
    uint256 totalSupply_,
    address owner_,
    address dest_
) external override returns (address token) {
    ...

    IToken(token).initialize(symbol_, name_, totalSupply_, owner_, dest_);
```

```
        ...
    }
```

2.In the TokenLocker contract, When the *addLock function is called to create new lock-up information, the parameter owner* will be used to set the owner role address of this lock. However, there is a lack of non-zero address check here. If the owner address is mistakenly set to the zero address, it could potentially lead to unexpected errors.

Code Location:

contracts/TokenLocker.sol#L127

```
    function _addLock(
        address token_,
        bool isLpToken_,
        address owner_,
        uint256 amount_,
        uint256 endTime_,
        uint256 cycle_,
        uint24 tgeBps_,
        uint24 cycleBps_,
        bytes32 feeNameHash_
    ) internal returns (uint256 lockId) {
        lockId = nextLockId;
        locks[lockId] = LockInfo({
            ...

            owner: owner_,

            ...
        });
        nextLockId++;
    }
```

3.In the UniV3LPLocker contract, When the *lock function is called to create new lock-up information, the parameter owner* will be used to set the owner role address of this lock. However, there is a lack of non-zero address check here. If the owner address is mistakenly set to the zero address, it could potentially lead to unexpected errors.

Code Location:

contracts/UniV3LPLocker.sol#L265

```
function _lock(
     INonfungiblePositionManager nftManager_,
     uint256 nftId_,
     address owner_,
     address collector_,
     address collectAddress_,
     uint256 endTime_,
     FeeStruct memory feeObj
 ) internal returns (uint256 lockId) {
     ...

     LockInfo memory newLock = LockInfo({
         ...

         owner: owner_,

         ...
     });

     ...
 }
```

Solution

It is recommended that the address non-zero checks should be added.

Status

Fixed; The project team responded: Allow users to set the lock owner to zero address, which means permanent lock

[N2] [Medium] Risk of front-running attack

Category: Authority Control Vulnerability Audit

Content

In the TokenTemplate contract, the devInit function is used to set up configurations necessary for token transfers, such as blacklists, whitelists, fee recipient addresses, and so on. However, due to the lack of a permission check in this function, it can be called by anyone. This leads to a potential front-running attack scenario: when a new token is created, an attacker can call this function to set these configurations before the owner role does. And since the function can only be called once, the owner would be unable to make any changes thereafter.

Code Location:

contracts/TokenTemplate.sol#L51-94

```
function devInit(
    address[] calldata blacks_,
    address[] calldata whites_,
    address[] calldata pool_,
    uint256 buyTax_,
    uint256 sellTax_,
    address taxReceiver_) external
{
    ...
}
```

Solution

It is suggested to add permission checks to the devInit function, restricting its invocation to the owner role only.

Status

Fixed

[N3] [Medium] Lack of check on the blacklist setting

Category: Design Logic Audit

Content

When calling the devInit function, you can set a list of blacklisted addresses for the token. Addresses on this blacklist will be prohibited from transferring or receiving the token. However, there is no check when adding blacklisted addresses to ensure that they cannot be the taxReceiver address or the dest_ address(which receives all tokens during initialization). If these addresses are mistakenly set to the blacklist, the token transfer function may not work properly.

Code Location:

contracts/TokenTemplate.sol#L65-70

```
function devInit(
    address[] calldata blacks_,
    address[] calldata whites_,
    address[] calldata pool_,
    uint256 buyTax_,
    uint256 sellTax_,
    address taxReceiver_) external
{
    ...
```

```
        // set blacklist
        for(uint256 i = blacks_.length; i > 0; ) {
            unchecked {
                i--;
            }
            blacklist[blacks_[i]] = true;
        }

        ...
    }
```

Solution

It's recommended to add a check when setting the blacklist to ensure that the taxReceiver address and the dest_ address, which receives all tokens during initialization, are not added to the blacklist.

Status

Fixed

[N4] [Suggestion] Missing the event records

Category: Others

Content

In the following contracts, the owner role can modify some sensitive parameters, but there are no event logs in these functions.

Code Location:

contracts/TokenTemplate.sol#L133-139

```
    function removeBlacklist() external onlyOwner {
        ...
    }

    function removeTax() external onlyOwner {
        ...
    }
```

contracts/UniV3LPLocker.sol#L136-146

```
    function updateFeeReceiver(address feeReceiver_) external onlyOwner {
        ...
```

```
    }

    function updateFeeSigner(address feeSigner_) external onlyOwner {
        ...
    }

    function addSupportedNftManager(address nftManager_) external onlyOwner {
        ...
    }
```

Solution

It is recommended to record events when sensitive parameters are modified for self-inspection or community review.

Status

Fixed

[N5] [Low] Missing scope limit

Category: Others

Content

1.In the TokenLocker and UniV3LPLocker contract, the owner role can set the fee information for specific tokens by calling the addOrUpdateFee function. However, there's no maximum limit set for the lockFee_ and lpFee_. If the value is set too high, it may lead to unexpected depletion of users' assets or even cause unintended errors.

Code Location:

contracts/TokenLocker.sol#L64

```
    function addOrUpdateFee(string memory name_, uint24 lpFee_, uint256 lockFee_,
address lockFeeToken_, bool isLp) public onlyOwner {
        bytes32 nameHash = keccak256(abi.encodePacked(name_));

        FeeStruct memory feeObj = FeeStruct(name_,  lockFee_, lockFeeToken_, lpFee_);
        fees[nameHash] = feeObj;

        ...
    }
```

2.In the TokenLocker contract, when calling the _vestingLock function, the `params.cycle` parameter will be used to set the time interval for linear unlocking. However, there's no maximum limit set for the `params.cycle`. If the value

is set too high, it may result in parts of the tokens during the unlocking process being calculated as 0 due to rounding.

Code Location:

contracts/TokenLocker.sol#L223

```
    function _vestingLock(VestingLockParams memory params, string memory feeName_)
internal returns (uint256 lockId) {
        ...

        lockId = _addLock(
            ...

            params.cycle,

            ...
        );

        ...
    }

    function _withdrawableTokens(
        LockInfo memory userLock
    ) internal view returns (uint256) {
        ...

        if (block.timestamp >= userLock.endTime) {
            currentTotal =
                (((block.timestamp - userLock.endTime) / userLock.cycle) *
                    cycleReleaseAmount) +
                tgeReleaseAmount;
        }

        ...
    }
```

Solution

It's recommended to set a maximum limit on the aforementioned key parameters.

Status

Fixed; The project team responded: the lockFee is an amount of lockFeeToken, it depends on the decimals of the

token, and the specific amount limit is difficult to determine. And the cycle is determined by the user without any scope limit.

[N6] [Low] Token compatibility issues

Category: Design Logic Audit

Content

In the TokenLocker contract, when calling the _createLock, _vestingLock and _updateLock functions to transfer tokens to the contract, the lock amount is not calculated based on the difference in the contract balance before and after the transfer. Instead, the _amount parameter passed in is directly used. Therefore, if the transferred tokens are reflective tokens(deflation/inflation type tokens), the actual lock amount will not be equal to the number of tokens transferred into the contract, resulting in a calculation error.

Code Location:

contracts/TokenLocker.sol

```
    function _createLock(
        address token_,
        string memory feeName_,
        address owner_,
        uint256 amount_,
        uint256 endTime_
    ) internal returns (uint256 lockId) {
        TransferHelper.safeTransferFrom(
            token_,
            _msgSender(),
            address(this),
            amount_
        );
        bytes32 nameHash = keccak256(abi.encodePacked(feeName_));
        (bool isLpToken_, uint256 newAmount) = _takeFee(token_, amount_, nameHash);

        ...
    }

    ...

    function _vestingLock(VestingLockParams memory params, string memory feeName_)
internal returns (uint256 lockId) {
        ...
```

```
        TransferHelper.safeTransferFrom(
            params.token,
            _msgSender(),
            address(this),
            params.amount
        );
        bytes32 nameHash = keccak256(abi.encodePacked(feeName_));
        (bool isLpToken, uint256 newAmount) = _takeFee(params.token, params.amount,
nameHash);

        ...
    }
    ...

    function _updateLock(
        uint256 lockId_,
        uint256 moreAmount_,
        uint256 newEndTime_
    ) internal {
        ...

        TransferHelper.safeTransferFrom(
            userLock.token,
            lockOwner,
            address(this),
            moreAmount_
        );
        (, uint256 newAmount) = _takeFee(userLock.token, moreAmount_,
userLock.feeNameHash);

        ...
    }
```

Solution

It is recommended to use the difference in the token balance in the contract before and after the user's transfer as the actual lock amount of the user, instead of directly using the amount parameter passed in.

Status

Fixed

[N7] [Suggestion] lockInfo.amount is not set to zero

Category: Design Logic Audit

Content

In the TokenLocker contract, when calling the _vestingUnlock function for linear unlocking, if all locked tokens have been fully claimed after this unlock, the value of lockInfo.amount will not be set to 0. Although the user won't be able to claim more tokens afterward, this can mislead the user or the frontend.

Code Location:

contracts/TokenLocker.sol#L395-409

```
function _vestingUnlock(LockInfo storage lockInfo) internal {
    ...

    if (newTotalUnlockAmount == lockInfo.amount) {
        tokenLocks[lockInfo.token].remove(lockInfo.lockId);
        if(lockInfo.isLpToken) {
            userLpLocks[lockInfo.owner].remove(lockInfo.lockId);
        } else {
            userNormalLocks[lockInfo.owner].remove(lockInfo.lockId);
        }
        emit OnUnlock(
            lockInfo.lockId,
            lockInfo.token,
            msg.sender,
            newTotalUnlockAmount,
            block.timestamp
        );
    }

    ...
}
```

Solution

It's recommended that after unlocking, if all locked tokens have been fully claimed, then the value of lockInfo.amount should be set to zero.

Status

Fixed

[N8] [Suggestion] Permission check repetition

Category: Authority Control Vulnerability Audit

Content

In the TokenLocker contract, the unlock function should only be callable by the owner address corresponding to the

specified lock. However, the permission check is already performed in the validLockOwner modifier, so there is no need to repeat the check within the function.

Code Location:

contracts/TokenLocker.sol#L355

```
    function unlock(
        uint256 lockId_
    ) external override validLockOwner(lockId_) nonReentrant {
        LockInfo storage lockInfo = locks[lockId_];
        require(lockInfo.owner == _msgSender(), "Not owner");

        ...
    }
```

Solution

It's recommended to remove the permission check within the function.

Status

Fixed

[N9] [Suggestion] Repeat adding the names of supported fee information

Category: Design Logic Audit

Content

In the TokenLocker contract, the owner role can call the addOrUpdateFee function to add or modify fee information. However, if the fee information has already been added, there is no need to repeatedly add the hash of the fee information's name to the lpSupportedFeeNames or tokenSupportedFeeNames arrays.

Code Location:

contracts/TokenLocker.sol#L72-76

```
    function addOrUpdateFee(string memory name_, uint24 lpFee_, uint256 lockFee_,
address lockFeeToken_, bool isLp) public onlyOwner {
        ...

        if(isLp) {
            lpSupportedFeeNames.add(nameHash);
        } else {
```

```
            tokenSupportedFeeNames.add(nameHash);
        }
    }
```

Solution

It's recommended to add a check to the addOrUpdateFee function; if the specified fee information's name hash already exists in the lpSupportedFeeNames or tokenSupportedFeeNames array, then there is no need to add it.

Status

Fixed

[N10] [Medium] Signature replay risk

Category: Replay Vulnerability

Content

In the UniV3LPLocker contract, the lockWithCustomFee function allows users to pass in custom fee information data, authenticated by the signer role's signature, for transaction fee deduction operations. However, since the hash of the signed message does not include an incrementable nonce value, and there is also no mechanism to mark and check used signatures after verification, this leads to anyone being able to replay previously used signatures with custom fee information, which is not intended behavior.

Code Location:

contracts/UniV3LPLocker.sol#L223-231

```
    function _verifySignature(
        FeeStruct memory fee,
        bytes memory signature
    ) internal view {
        bytes32 messageHash = keccak256(abi.encodePacked(fee.name, fee.lpFee,
fee.collectFee, fee.lockFee, fee.lockFeeToken));
        bytes32 prefixedHash = MessageHashUtils.toEthSignedMessageHash(messageHash);
        address signer = ECDSA.recover(prefixedHash, signature);
        require(signer == customFeeSigner, "FeeSigner not allowed");
    }
```

Solution

It’s recommended to implement a flag in the lockWithCustomFee function to check if the signature has been used

before, and to add a nonce that increments each time when computing the hash.

Status

Fixed

[N11] [Medium] Missing chainID check in the signature verification

Category: Replay Vulnerability

Content

In the UniV3LPLocker contract, the lockWithCustomFee function allows users to pass in custom fee information data, authenticated by the signer role's signature, for transaction fee deduction operations. However, when hashing the data that needs to be checked for the signature, the chainId is not included in the hash. This means that if the project is deployed on multiple chains, the signature may be maliciously replayed by an attacker on another chain, which may result in unexpected incoming fee information.

Code Location:

contracts/UniV3LPLocker.sol#L227

```
    function _verifySignature(
        FeeStruct memory fee,
        bytes memory signature
    ) internal view {
        bytes32 messageHash = keccak256(abi.encodePacked(fee.name, fee.lpFee,
fee.collectFee, fee.lockFee, fee.lockFeeToken));

        ...
    }
```

Solution

If the project plans to be deployed on multiple chains, it’s recommended to add the chainId in the hash calculation of the signature message.

Status

Fixed

[N12] [Low] Potential Sandwich Attack Risk

Category: Reordering Vulnerability

Content

In the UniV3LPLocker contract, when calling the _lock function to perform a lock operation, if lpFee is greater than 0, an external call is made to the nftManager's decreaseLiquidity function to remove some liquidity and transfer the tokens to the feeReceiver address. However, when calling the decreaseLiquidity function, the amount0Min and amount1Min parameters passed are both 0, which could expose the tokens to sandwich attack arbitrage and result in losses.

Code Location:

contracts/UniV3LPLocker.sol#L257

```
    function _lock(
        INonfungiblePositionManager nftManager_,
        uint256 nftId_,
        address owner_,
        address collector_,
        address collectAddress_,
        uint256 endTime_,
        FeeStruct memory feeObj
    ) internal returns (uint256 lockId) {
        ...

        if (feeObj.lpFee > 0) {
            uint128 liquidity = _getLiquidity(nftManager_, nftId_);
            nftManager_.decreaseLiquidity(
                INonfungiblePositionManager.DecreaseLiquidityParams(nftId_,
uint128(liquidity * feeObj.lpFee / FEE_DENOMINATOR),
                0, 0, block.timestamp));
            nftManager_.collect(INonfungiblePositionManager.CollectParams(nftId_,
feeReceiver, type(uint128).max, type(uint128).max));
        }

        ...
    }
```

Solution

It's recommended to add parameters for the minimum output amounts when removing liquidity in the _lock function instead of setting them directly to 0 to control slippage.

Status

Acknowledged

[N13] [Suggestion] Redundant code

Category: Others

Content

In the UniV3LPLocker contract, when performing a lock operation, a collectAddress parameter will be added to the lock information. However, this collectAddress has no involvement in the business logic of the entire contract.

Code Location:

contracts/UniV3LPLocker.sol

```
    function _lock(
        INonfungiblePositionManager nftManager_,
        uint256 nftId_,
        address owner_,
        address collector_,
        address collectAddress_,
        uint256 endTime_,
        FeeStruct memory feeObj
    ) internal returns (uint256 lockId) {
        ...

        LockInfo memory newLock = LockInfo({
            ...

            collectAddress: collectAddress_,

            ...
        });

        ...
    }
```

Solution

If collectAddress is not needed in the subsequent expected design, it is recommended to delete it.

Status

Fixed

[N14] [Information] No handling fee will be settled first when transferring the lock permission

Category: Design Logic Audit

Content

In the UniV3LPLocker contract, the acceptLock function is used for the newly set pendingOwner to accept the management rights of the specified lock. However, when transferring the permissions, the liquidity earnings of the previous owner are not settled first. This allows the newly designated owner to claim the liquidity earnings that belonged to the previous owner.

Code Location:

contracts/UniV3LPLocker.sol#L296-310

```
    function acceptLock(uint256 lockId_) external {
        ...
    }
```

Solution

If this is as per the expected design, no modifications are needed; otherwise, it's best to call the collect function to settle the previous owner's liquidity earnings before transferring management rights.

Status

Acknowledged; The project team responded: Known issues. Whether a user needs to settle before transferring ownership depends on themselves, they can call collect method before transfer.

[N15] [Information] Blacklist and whitelist cannot be changed after setting

Category: Design Logic Audit

Content

In the TokenTemplate contract, After the devInit function is called, the token's blacklist address list and whitelist address list cannot be added to, modified, or deleted from.

Code Location:

contracts/TokenTemplate.sol#L65-78

```
    function devInit(
        address[] calldata blacks_,
```

```
        address[] calldata whites_,
        address[] calldata pool_,
        uint256 buyTax_,
        uint256 sellTax_,
        address taxReceiver_) external
    {
        ...

        // set blacklist
        for(uint256 i = blacks_.length; i > 0; ) {
            unchecked {
                i--;
            }
            blacklist[blacks_[i]] = true;
        }

        // set whitelist
        for(uint256 i = whites_.length; i > 0; ) {
            unchecked {
                i--;
            }
            whitelist[whites_[i]] = true;
        }

        ...
    }
```

Solution

N/A

Status

Acknowledged; The project team responded: Known issues. Restricting the permissions of token issuers.

[N16] [Medium] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

1.In the SafeTokenFactory contract, the owner role can call the updateTemplates function to set the address of the template contract. If the private key for the core role is lost, it may cause the token's template contract to be set to a malicious address.

Code Location:

contracts/SafeTokenFactory.sol#L58-64

```
    function updateTemplates(
        uint256 tempKey_,
        address templete_
    ) external onlyOwner {
        ...
    }
```

2.In the TokenLocker contract, the owner role can call the addOrUpdateFee and updateFeeReceiver function to set the fee information and the feeReceiver address. If the private key for the core role is lost, it may result in a loss of funds for the contract.

Code Location:

contracts/TokenLocker.sol#L61-83

```
    function addOrUpdateFee(string memory name_, uint24 lpFee_, uint256 lockFee_,
address lockFeeToken_, bool isLp) public onlyOwner {
        ...
    }

    function updateFeeReceiver(address feeReceiver_) external onlyOwner {
        ...
    }
```

3.In the UniV3LPLocker contract, the owner role can call the addOrUpdateFee, removeFee, updateFeeSigner and addSupportedNftManager to set the fee information, the feeReceiver, the customFeeSigner and the nftManagers list. In addition, the owner can also call the adminRefundEth and adminRefundERC20 functions to transfer tokens within the contract.

Code Location:

contracts/UniV3LPLocker.sol

```
    function addOrUpdateFee(string memory name_, uint256 lpFee_, uint256 collectFee_,
uint256 lockFee_, address lockFeeToken_) public onlyOwner {
        ...
    }
```

```
    function removeFee(string memory name_) external onlyOwner {
        ...
    }

    function updateFeeReceiver(address feeReceiver_) external onlyOwner {
        ...
    }

    function updateFeeSigner(address feeSigner_) external onlyOwner {
        ...
    }

    function addSupportedNftManager(address nftManager_) external onlyOwner {
        ...
    }

    function adminRefundEth (uint256 amount_, address payable receiver_) external
onlyOwner nonReentrant {
        ...
    }

    function adminRefundERC20 (address token_, address receiver_, uint256 amount_)
external onlyOwner nonReentrant {
        ...
    }
```

Solution

In the short term, during the early stages of the project, the protocol may need to frequently set various parameters to ensure the stable operation of the protocol. Therefore, transferring the owner ownership to a multisig management can effectively solve the single-point risk, but it cannot mitigate the excessive privilege risk. In the long run, after the protocol stabilizes, transferring the owner ownership to community governance and executing through a timelock can effectively mitigate the excessive privilege risk and increase the community users' trust in the protocol.

Status

Acknowledged; The project team responded: After the protocol runs stably for a period of time, we will consider adding community governance and timelock.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002409180001	SlowMist Security Team	2024.09.12 - 2024.09.18	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 5 medium risk, 3 low risk, 6 suggestion vulnerabilities and 2 information. All the findings were fixed and acknowledged. Since the project has not yet been deployed to the mainnet and the permissions of the core roles have not yet been transferred, the risk level reported is temporarily medium.

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