SMART CONTRACT AUDIT REPORT

for

HOPE Ecosystem

Prepared By: Xiaomi Huang

PeckShield

April 25, 2023

Document Properties

Client	HOPE Ecosystem
Title	Smart Contract Audit Report
Target	HOPE Ecosystem
Version	1.2
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.2	April 25, 2022	Luck Hu	Post Release #2
1.1	April 19, 2022	Luck Hu	Post Release #1
1.0	April 15, 2022	Luck Hu	Final Release
1.0-rc	April 13, 2023	Luck Hu	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the HOPE Ecosystem, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About HOPE Ecosystem

The HOPE Ecosystem and its native stablecoin $HOPE, aim to provide frictionless and transparent next-generation financial infrastructure and services accessible to everyone. The HOPE Ecosystem provides a comprehensive set of use cases for $HOPE, including swap, lending, custody, clearing, and settlement, while incentivizing users to participate in the ecosystem and community governance through $LT. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of HOPE Ecosystem

Item	Description
Name	HOPE Ecosystem
Website	https://hope.money/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 25, 2023

In the following, we show the Git repository of reviewed files and the commit hash values used in this audit.

- https://github.com/Light-Ecosystem/light-dao/tree/audit_2 (8783998)

- https://github.com/Light-Ecosystem/light-lib/tree/audit (123a60b)
- https://github.com/Light-Ecosystem/light-permit2/tree/audit (a5ffec8)
- https://github.com/Light-Ecosystem/swap-core/tree/audit (660f06)
- https://github.com/Light-Ecosystem/swap-periphery/tree/audit (7ae41e7)
- https://github.com/Light-Ecosystem/light-vest-escrow/tree/audit (a8e6530)

And this is the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/Light-Ecosystem/light-dao/tree/audit_2 (562847a)
- https://github.com/Light-Ecosystem/light-lib/tree/audit (9a9f28f)
- https://github.com/Light-Ecosystem/light-permit2/tree/audit (a5ffec8)
- https://github.com/Light-Ecosystem/swap-core/tree/audit (660f06)
- https://github.com/Light-Ecosystem/swap-periphery/tree/audit (7ae41e7)
- https://github.com/Light-Ecosystem/light-vest-escrow/tree/audit (187ab54)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the HOPE protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■■■
Low	2	■■
Informational	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-002	Medium	Revised Selection of bestRouter in HopeSwapBurner::burn()	Business Logic	Fixed
PVE-003	Low	Revisited Slippage Control in SwapFeeToVault	Time and State	Mitigated
PVE-004	Medium	Trust Issue on Admin Keys	Security Features	Mitigated
PVE-005	Medium	Proper Claim of Fee in claimableTokens()	Business Logic	Fixed

Beside the identified issues, we note that the staking support assumes the staked tokens are not deflationary. Also, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-628 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transfer()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.1: USDT::transfer()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In current implementation, we show the `_withdraw()` routine in the `PoolGauge` contract. If the `USDT` token is supported as `lpToken`, the unsafe version of `IERC20Metadata(lpToken).transfer(msg.sender, _value)` (line 186) may success with no return value. However, the following validation expects it to return `true` (line 187). As a result, the transaction reverts.

```
169   function _withdraw(uint256 _value, bool _claimRewards_) private {
170     _checkpoint(msg.sender);

172     if (_value != 0) {
173         bool isRewards = rewardCount != 0;
174         uint256 _totalSupply = totalSupply;
175         if (isRewards) {
176             _checkpointRewards(msg.sender, _totalSupply, _claimRewards_, address(0));
177         }

179         _totalSupply -= _value;
180         uint256 newBalance = balanceOf[msg.sender] - _value;
181         balanceOf[msg.sender] = newBalance;
182         totalSupply = _totalSupply;

184         _updateLiquidityLimit(msg.sender, newBalance, _totalSupply);

186         bool success = IERC20Metadata(lpToken).transfer(msg.sender, _value);
187         require(success, "TRANSFER FAILED");
188     }

190     emit Withdraw(msg.sender, _value);
191     emit Transfer(msg.sender, address(0), _value);
192   }
```

Listing 3.2: `PoolGauge::_withdraw()`

In the meantime, we also suggest to use the safe-version of `transfer()/transferFrom()` in other related routines, including `HopeSwapBurner::burn()` and `SwapFeeToVault::_burn()`, etc.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been fixed in these commits: `965a739` and `9a9f28f`.

3.2 Revised Selection of bestRouter in HopeSwapBurner::burn()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `HopeSwapBurner/UnderlyingBurner`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `LightDAO` governance protocol, the `HopeSwapBurner` contract is designed to convert the received fees to `HOPE`. It has a list of candidate routers which can be used to complete the swap. In order to receive the most `HOPE` from the swap, it makes queries to current routers and selects the best one to complete the swap. While reviewing the selection of the best router, we notice it does not use the correct return value from the query.

To elaborate, we show below the code snippet of the `HopeSwapBurner::burn()` routine, which is designed to convert the given fee token, i.e., `token`, to `HOPE`. It asks queries for the swap from each of the candidate routers and selects the best one which can offer the most `HOPE`. The query is carried out by calling the `getAmountsOut()` routine of each router (line 66), which returns an array of amounts, i.e., `expected`. The `expected` array records the amount of the target token it can receive from each step of the swap path. Specially, the first element in the array (`expected[0]`) is the input token amount, i.e., `spendAmount`, and the last element in the array (`expected[1]`) is the amount of the target token, i.e., `HOPE`.

Based on this, it shall use the `expected[1]` as the query result to choose the best router, not the `expected[0]` (line 67).

```
46     function burn(address to, IERC20 token, uint amount, uint amountOutMin) external {
47         require(msg.sender == feeVault, "LSB04");
48 
49         if (token == HOPE) {
50             require(token.transferFrom(msg.sender, to, amount), "LSB00");
51             return;
52         }
53 
54         uint256 balanceBefore = token.balanceOf(address(this));
55         require(token.transferFrom(msg.sender, address(this), amount), "LSB01");
56         uint256 balanceAfter = token.balanceOf(address(this));
57         uint256 spendAmount = balanceAfter - balanceBefore;
58 
59         ISwapRouter bestRouter = routers[0];
60         uint bestExpected = 0;
61         address[] memory path = new address[](2);
```

```
        path[0] = address(token);
        path[1] = address(HOPE);

        for (uint i = 0; i < routers.length; i++) {
            uint[] memory expected = routers[i].getAmountsOut(spendAmount, path);
            if (expected[0] > bestExpected) {
                bestExpected = expected[0];
                bestRouter = routers[i];
            }
        }

        require(bestExpected >= amountOutMin, "LSB02");
        if (!approved[bestRouter][token]) {...}

        bestRouter.swapExactTokensForTokens(spendAmount, 0, path, to, block.timestamp);
    }
```

Listing 3.3: HopeSwapBurner::burn()

Note the same issue is also applicable to the `UnderlyingBurner::burn()` routine.

Recommendation Revise the above mentioned `burn()` routine and use `expected[1]` as the query result to select the best router.

Status This issue has been fixed in this commit: `21f2c9f`.

3.3 Revisited Slippage Control in SwapFeeToVault

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SwapFeeToVault`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

In the `LightDAO` governance protocol, the `SwapFeeToVault` contract is designed to withdraw the admin fees from the `SwapPair` and convert the fees to its stablecoin `HOPE`. Our analysis shows that the current implementation lacks an effective slippage control that occurs in the conversion from the fees to `HOPE`.

To elaborate, we show below the related `SwapFeeToVault::burn()` routine, which is used to convert the given fee token to `HOPE` by calling the `burner.burn()` (line 54). In order to protect the conversion from possible loss, it requires the caller to provide the minimum acceptable amount of `HOPE`, i.e., `amountOutMin`. However, we notice there is no access control for the `SwapFeeToVault::burn()` routine, and the caller can provide any value for the `amountOutMin` parameter. As a result, if the given `amountOutMin` is too small, there is no effective slippage control for the conversion.

Based on this, we suggest to add proper access control to the `SwapFeeToVault::burn()` routine, so it can rely on the privileged caller to provide a reasonable `amountOutMin`.

Note the same issue is also applicable to the `SwapFeeToVault::burnMany()` routine.

```
    function _burn(IERC20 token, uint amountOutMin) internal {
        uint256 amount = token.balanceOf(address(this));
        // user choose to not burn token if not profitable
        if (amount > 0) {
            IBurner burner = burnerManager.burners(address(token));
            require(token.approve(address(burner), amount), "SFTV00");
            require(burner != IBurner(address(0)), "SFTV01");
            burner.burn(underlyingBurner, token, amount, amountOutMin);
        }
    }

    function burn(IERC20 token, uint amountOutMin) external whenNotPaused {
        require(msg.sender == tx.origin, "SFTV02");
        _burn(token, amountOutMin);
    }
```

Listing 3.4: `SwapFeeToVault::burn()`

What is more, the `SwapFeeToVault` contract provides an interface, i.e., `withdrawAdminFee()`, to withdraw the admin fees from the supported `SwapPair`. We notice the withdrawal of the admin fees is directly carried out by calling the `pair.burn()` routine to remove liquidity (line 37), and essentially there is no effective restriction on the received tokens amounts. As a result, if it is removing liquidity from an imbalanced pair, it cannot receive the desired amounts of tokens. Based on this, we suggest to add a proper validation on the received tokens and and ensure it can receive the desired amounts of tokens from the pair.

Note the same issue is also applicable to the `withdrawMany()` routine.

```
31     function withdrawAdminFee(address pool) external whenNotPaused {
32         SwapPair pair = SwapPair(pool);
33         pair.mintFee();
34         uint256 tokenPBalance = SwapPair(pool).balanceOf(address(this));
35         if (tokenPBalance > 0) {
36             pair.transfer(address(pair), tokenPBalance);
37             pair.burn(address(this));
38         }
39     }
```

Listing 3.5: `SwapFeeToVault::withdrawAdminFee()`

Recommendation Revisit the slippage control in the `SwapFeeToVault` contract to protect the admin fees from possible slippage loss.

Status The issue has been mitigated as the team add access control to the above mentioned routines in this commit: `0c5e2a5`, and only allow privileged role to call these routines.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the LightDAO governance protocol, there is a special owner account that plays a critical role in governing and regulating the protocol-wide operations (e.g., change gauge weight, recover HOPE from the fee distributors). Our analysis shows that the owner account needs to be scrutinized. In the following, we use the GaugeController contract as an example and show the representative functions potentially affected by the privileges of the owner account.

Specifically, the privileged functions in GaugeController allow for the owner to add a gauge type with any weight, add a gauge with any weight to a gauge type, change the weight of a gauge type, change the weight of a gauge, etc. Note the gauge type weight and the gauge weight can directly impact the LT rewards distribution in the gauge and the admin fees distribution in the gauge fee distributor.

```
    /**
     * @notice Add gauge ‘addr‘ of type ‘gaugeType‘ with weight ‘weight‘
     * @param addr Gauge address
     * @param gaugeType Gauge type
     * @param weight Gauge weight
     */
    function addGauge(address addr, int128 gaugeType, uint256 weight) external override
        onlyOwner {
        require(gaugeType >= 0 && gaugeType < nGaugeTypes, "GC001");
        require(_gaugeTypes[addr] == 0, "GC002");

        int128 n = nGauge;
        nGauge = n + 1;
        gauges[_int128ToUint256(n)] = addr;

        _gaugeTypes[addr] = gaugeType + 1;
        uint256 nextTime = LibTime.timesRoundedByWeek(block.timestamp + _WEEK);

        if (weight > 0) {
            uint256 _typeWeight = _getTypeWeight(gaugeType);
            uint256 _oldSum = _getSum(gaugeType);
            uint256 _oldTotal = _getTotal();

            pointsSum[gaugeType][nextTime].bias = weight + _oldSum;
            timeSum[_int128ToUint256(gaugeType)] = nextTime;
```

```
            pointsTotal[nextTime] = _oldTotal + _typeWeight * weight;
            timeTotal = nextTime;

            pointsWeight[addr][nextTime].bias = weight;
        }

        if (timeSum[_int128ToUint256(gaugeType)] == 0) {
            timeSum[_int128ToUint256(gaugeType)] = nextTime;
        }
        timeWeight[addr] = nextTime;

        emit NewGauge(addr, gaugeType, weight);
    }

    /**
    * @notice Add gauge type with name `_name` and weight `weight`
    * @dev only owner call
    * @param _name Name of gauge type
    * @param weight Weight of gauge type
    */
    function addType(string memory _name, uint256 weight) external override onlyOwner {
        int128 typeId = nGaugeTypes;
        gaugeTypeNames[typeId] = _name;
        nGaugeTypes = typeId + 1;
        if (weight != 0) {
            _changeTypeWeight(typeId, weight);
        }
        emit AddType(_name, typeId);
    }

    /**
     * @notice Change gauge type `typeId` weight to `weight`
     * @dev only owner call
     * @param typeId Gauge type id
     * @param weight New Gauge weight
     */
    function changeTypeWeight(int128 typeId, uint256 weight) external override onlyOwner {
        _changeTypeWeight(typeId, weight);
    }

    /**
     * @notice Change weight of gauge `addr` to `weight`
     * @param gaugeAddress `Gauge` contract address
     * @param weight New Gauge weight
     */
    function changeGaugeWeight(address gaugeAddress, uint256 weight) external override onlyOwner {
        int128 gaugeType = _gaugeTypes[gaugeAddress] - 1;
        require(gaugeType >= 0, "GC000");
        _changeGaugeWeight(gaugeAddress, weight);
```

```
    }
```

Listing 3.6: Example Privileged Operations in `GaugeController`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged account may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated as the team confirm they are using a multi-sig account as the `owner`, and will transfer the `owner` to the community in future.

3.5 Proper Claim of Fee in claimableTokens()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `GaugeFeeDistributor/FeeDistributor`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `LightDAO` governance protocol, the admin fees withdrawn from the `SwapPair` are transferred to the `GaugeFeeDistributor` and the `FeeDistributor` in the format of `HOPE`. The `HOPE` can be claimed by the community for their holding of `veLT` and the voting for the gauges. While examining the claiming of the `HOPE` rewards via the `GaugeFeeDistributor::claimableTokens()` routine, we notice it updates user's claim state but does not transfer the claimable rewards to the user.

In the following, we show the code snippets of the `GaugeFeeDistributor::claimableTokens()/_claim()` routines. As the name indicates, the `GaugeFeeDistributor::claimableTokens()` routine is intended to be a helper function for the user to query his/her claimable `HOPE`. It invokes the `GaugeFeeDistributor::_claim()` routine (line 287) to count the claimable amount. In the `GaugeFeeDistributor::_claim()` routine, it moves the user's claimed epoch and time cursor to the latest before returning the claimable amount (lines 267 – 268). However, the corresponding `HOPE` is not transferred to the user.

Based on this, we suggest to transfer the claimable `HOPE` to the user or don't update the claim state for the user in the `GaugeFeeDistributor::claimableTokens()` routine.

```
    function _claim(address gauge, address addr, uint256 _lastTokenTime) internal
        returns (uint256) {
        ClaimParam memory param;

        ///Minimal userEpoch is 0 (if user had no point)
        param.userEpoch = 0;
        param.toDistribute = 0;

        param.maxUserEpoch = IGaugeController(gaugeController).lastVoteVeLtPointEpoch(
            addr, gauge);
        uint256 _startTime = startTime;
        if (param.maxUserEpoch == 0) {...}

        param.weekCursor = timeCursorOf[gauge][addr];
        if (param.weekCursor == 0) {
            /// Need to do the initial binary search
            param.userEpoch = _findTimestampUserEpoch(gauge, addr, _startTime, param.
                maxUserEpoch);
        } else {
            param.userEpoch = userEpochOf[gauge][addr];
        }
        if (param.userEpoch == 0) {
            param.userEpoch = 1;
        }

        param.userPoint = IGaugeController(gaugeController).voteVeLtPointHistory(addr,
            gauge, param.userEpoch);
        if (param.weekCursor == 0) {
            param.weekCursor = LibTime.timesRoundedByWeek(param.userPoint.ts + WEEK - 1)
                ;
        }

        if (param.weekCursor >= _lastTokenTime) {
            return 0;
        }

        if (param.weekCursor < _startTime) {
            param.weekCursor = _startTime;
        }
        param.oldUserPoint = Point({bias: 0, slope: 0, ts: 0, blk: 0});

        /// Iterate over weeks
        for (int i = 0; i < 50; i++) {...}

        param.userEpoch = Math.min(param.maxUserEpoch, param.userEpoch - 1);
        userEpochOf[gauge][addr] = param.userEpoch;
        timeCursorOf[gauge][addr] = param.weekCursor;

        emit Claimed(gauge, addr, param.toDistribute, param.userEpoch, param.
```

```
            maxUserEpoch);

        return param.toDistribute;
    }

    function claimableTokens(address gauge, address _addr) external whenNotPaused
        returns (uint256) {
        if (_addr == address(0)) {
            _addr = msg.sender;
        }
        uint256 _lastTokenTime = lastTokenTime;
        if (canCheckpointToken && (block.timestamp > _lastTokenTime +
            TOKEN_CHECKPOINT_DEADLINE)) {
            _checkpointToken();
            _lastTokenTime = block.timestamp;
        }

        _lastTokenTime = LibTime.timesRoundedByWeek(_lastTokenTime);
        IGaugeController(gaugeController).checkpointGauge(gauge);
        return _claim(gauge, _addr, _lastTokenTime);
    }
```

Listing 3.7: GaugeFeeDistributor::claimableTokens()

Recommendation Revisit the above `GaugeFeeDistributor::claimableTokens()` routine and properly transfer the claimed `HOPE` to the user or do not update user's claim state.

Status This issue has been fixed in this commit: `019f7c6`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `HOPE Ecosystem`. The `HOPE Ecosystem` and its native stablecoin `$HOPE`, aim to provide frictionless and transparent next-generation financial infrastructure and services accessible to everyone. The `HOPE Ecosystem` provides a comprehensive set of use cases for `$HOPE`, including swap, lending, custody, clearing, and settlement, while incentivizing users to participate in the ecosystem and community governance through `$LT`. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.