SMART CONTRACT AUDIT REPORT

for

LineaBank

Prepared By: Xiaomi Huang

PeckShield

July 17, 2023

Document Properties

Client	LineaBank
Title	Smart Contract Audit Report
Target	LineaBank
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 17, 2023	Luck Hu	Final Release
1.0-rc1	July 16, 2023	Luck Hu	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the LineaBank protocol, 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 audited protocol 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 LineaBank

LineaBank is in essence a lending protocol built on the Linea (ConsenSys zkEVM). The protocol gives users full control over their funds and offers competitive interest rates through a decentralized market that eliminates intermediaries. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of LineaBank

Item	Description
Name	LineaBank
Website	https://lineabank.finance
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 17, 2023

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

- https://github.com/lineabank/lineabank (8c4274d)

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

- https://github.com/lineabank/lineabank (717267a)

1.2 About PeckShield

PeckShield Inc. [7] 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).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- 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.

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 classified into four categories accordingly, 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

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

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) [5], which is a community-developed list of software weakness types to 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.

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 `LineaBank` implementation. 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	2	■■
Low	0	
Undetermined	0	
Total	4	

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 that 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 2 high-severity vulnerabilities and 2 medium-severity vulnerabilities.

Table 2.1: Key LineaBank Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Lack rewardStored Reset in SaleLabOverflowFarm::harvestOverflowReward()	Business Logic	Fixed
PVE-002	High	Incorrect Claiming Logic in RebateDistributor::claimAdminRebates()	Business Logic	Fixed
PVE-003	Medium	Revised Borrow/Supply Value Calculation in Liquidation::_getTargetMarkets()	Business Logic	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Beside the identified issues, 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 Lack rewardStored Reset in SaleLabOverflowFarm::harvestOverflowReward()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: SaleLabOverflowFarm
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the LineaBank protocol, the SaleLabOverflowFarm contract performs as an IDO contract which collects users deposit of the raising tokens and offers users with the offering tokens. Meanwhile, the user can earn the reward tokens from the IDO. While reviewing the harvest logic of the reward tokens, we notice an issue that a user may repeat the harvest operations to drain all the reward tokens in the contract.

To elaborate, we show below the code snippet of the SaleLabOverflowFarm::harvestOverflowReward() function. As the name indicates, it is used by the user to harvest the rewards. It basically calculates the new pending rewards (line 168), adds the stored rewards (line 169), i.e., rewardStored[msg.sender], and then transfers all the pending rewards to the user (line 171).

However, it comes to our attention that it doesn't reset the rewardStored[msg.sender] after the harvest. As a result, the user can repeatedly harvest to drain all the rewards in the contract. Our analysis shows that it should reset the rewardStored[msg.sender] after the harvest.

```
161     function harvestOverflowReward() external override nonReentrant {
162         require(block.timestamp > harvestTime(), "not harvest time");
163         UserInfo storage user = userInfo[msg.sender];
164         _updatePool();
165 
166         uint256 pending = 0;
167         if (user.amount > 0) {
```

```
            pending = user.amount.mul(accTokenPerShare).div(1e18).sub(user.rewardDebt);
            pending = pending.add(rewardStored[msg.sender]);
            if (pending > 0) {
                address(rewardToken).safeTransfer(msg.sender, pending);
            }
        }
        user.rewardDebt = user.amount.mul(accTokenPerShare).div(1e18);
    }
```

Listing 3.1: `SaleLabOverflowFarm::harvestOverflowReward()`

Recommendation Revisit the `SaleLabOverflowFarm::harvestOverflowReward()` function to reset the `rewardStored[msg.sender]` after the harvest.

Status This issue has been fixed in the following commit: 717267a0.

3.2 Incorrect Claiming Logic in RebateDistributor::claimAdminRebates()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: `RebateDistributor`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `LineaBank` protocol, the `RebateDistributor` contract provides functions for the keeper to claim admin rebates and the users to claim users rebates. While reviewing the rebate-claiming logic, we notice it calls a wrong internal routine to claim the admin rebates, and the claimed market fees are locked in the contract.

To elaborate, we show below the related code snippets of the `claimAdminRebates()/accruedAdminRebate()` routines. As the name indicates, the first one is used by the keeper to claim the admin rebates. Our analysis shows that it calls the `RebateDistributor::accruedRebates()` routine (line 162) to calculate the rebates for the admin, which is designed to calculate the rebates for normal users. As a result, it claims wrong rebates for the admin. To fix, it needs to call the `RebateDistributor::accruedAdminRebate()` routine to calculate the rebates for the admin.

Moreover, in the `RebateDistributor::claimAdminRebates()` routine, it updates the timestamp of the last claimed check point for the user (line 164), i.e., `userCheckpoint[msg.sender]`. Our analysis shows that it should update the timestamp of the last claimed check point for the admin, i.e., `adminCheckpoint`.

```
function claimAdminRebates()
    external
    override
    nonReentrant
    onlyKeeper
    returns (uint256 addtionalLabAmount, uint256[] memory marketFees)
  {
    (, addtionalLabAmount, marketFees) = accruedRebates(msg.sender);
    Constant.RebateCheckpoint memory lastCheckpoint = rebateCheckpoints[rebateCheckpoints.length - 1];
    userCheckpoint[msg.sender] = _truncateTimestamp(lastCheckpoint.timestamp.sub(REBATE_CYCLE));

    address(lab).safeTransfer(msg.sender, addtionalLabAmount);
  }

  function accruedAdminRebate() public view returns (uint256 additionalLabAmount, uint256[] memory marketFees) {
    Constant.RebateCheckpoint memory lastCheckpoint = rebateCheckpoints[rebateCheckpoints.length - 1];
    address[] memory markets = core.allMarkets();
    marketFees = new uint256[](markets.length);

    for (
      uint256 nextTimestamp = _truncateTimestamp(adminCheckpoint).add(REBATE_CYCLE);
      nextTimestamp <= lastCheckpoint.timestamp.sub(REBATE_CYCLE);
      nextTimestamp = nextTimestamp.add(REBATE_CYCLE)
    ) {
      uint256 checkpointIdx = _getCheckpointIdxAt(nextTimestamp);
      Constant.RebateCheckpoint storage currentCheckpoint = rebateCheckpoints[checkpointIdx];
      additionalLabAmount = additionalLabAmount.add(
        currentCheckpoint.additionalLabAmount.mul(currentCheckpoint.adminFeeRate).div(1e18)
      );

      for (uint256 i = 0; i < markets.length; i++) {
        if (currentCheckpoint.marketFees[markets[i]] > 0) {
          marketFees[i] = marketFees[i].add(
            currentCheckpoint.marketFees[markets[i]].mul(currentCheckpoint.adminFeeRate).div(1e18)
          );
        }
      }
    }
  }
```

Listing 3.2: `RebateDistributor::claimAdminRebates()`

What's more, the rebates are composed of `LAB` and market fees. The market fees are pulled into the contract via the `RebateDistributor::addMarketUTokenToRebatePool()` routine (as the code shown below). However, in the `RebateDistributor::claimAdminRebates()` routine, we notice it only transfers the rebates of LAB to the caller, the rebates of market fees are not transferred to the caller. As a result, the market fees are locked in the contract.

Note the same issue is also applicable to the RebateDistributor::claimRebates() routine, where the claimed market fees are not transferred to the user.

```
    function addMarketUTokenToRebatePool(address lToken, uint256 uAmount) external
        payable override nonReentrant {
        Constant.RebateCheckpoint storage lastCheckpoint = rebateCheckpoints[
            rebateCheckpoints.length - 1];
        address underlying = ILToken(lToken).underlying();

        if (underlying == ETH && msg.value > 0) {
          lastCheckpoint.marketFees[lToken] = lastCheckpoint.marketFees[lToken].add(msg.
              value);
        } else if (underlying != ETH) {
          address(underlying).safeTransferFrom(msg.sender, address(this), uAmount);
          lastCheckpoint.marketFees[lToken] = lastCheckpoint.marketFees[lToken].add(
              uAmount);
        }
      }
```

Listing 3.3: RebateDistributor::addMarketUTokenToRebatePool()

Recommendation Revisit the RebateDistributor::claimAdminRebates() routine to call the RebateDistributor::accruedAdminRebate() routine to calculate the rebates for the admin and properly transfer the market fees to the caller.

Status This issue has been fixed in the following commit: 717267a0, and the team confirmed that the rebates of market fees are not available now and the RebateDistributor::addMarketUTokenToRebatePool() routine will not be triggered currently.

3.3 Revised Borrow/Supply Value Calculation in Liquidation::_getTargetMarkets()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Liquidation
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the LineaBank protocol, the Liquidation contract performs as a liquidation bot to liquidate the borrower's debt and send the earns to the rebate distributor. Specially, it provides an autoLiquidate

() function that chooses the market with the maximum borrow value to liquidate and the market with the maximum supply value in return. While examining the calculation of the borrow/supply values in a market, we notice it does not take the token decimal into consideration.

In the following, we show the code snippet of the `Liquidation::_getTargetMarkets()` routine, which is used to choose the market with the maximum borrow value and the market with the maximum supply value. For each market, the borrow/supply values are calculated per the borrow/supply balances of the borrower and the underlying prices (lines 223 – 224).

However, it comes to our attention that it does not take the underlying token decimal into consideration. As a result, it does not remove the underlying token decimal from the calculated values and the chosen markets are not accurate. With that, we suggest to remove the underlying token decimal from the calculated values.

```
216     function _getTargetMarkets(address account) private view returns (address gTokenBorrowed, address gTokenCollateral) {
217         uint256 maxSupplied;
218         uint256 maxBorrowed;
219         address[] memory markets = core.marketListOf(account);
220         uint256[] memory prices = priceCalculator.getUnderlyingPrices(markets);
221
222         for (uint256 i = 0; i < markets.length; i++) {
223           uint256 borrowValue = ILToken(markets[i]).borrowBalanceOf(account).mul(prices[i]).div(1e18);
224           uint256 supplyValue = ILToken(markets[i]).underlyingBalanceOf(account).mul(prices[i]).div(1e18);
225
226           if (borrowValue > 0 && borrowValue > maxBorrowed) {
227             maxBorrowed = borrowValue;
228             gTokenBorrowed = markets[i];
229           }
230
231           uint256 collateralFactor = core.marketInfoOf(markets[i]).collateralFactor;
232           if (collateralFactor > 0 && supplyValue > 0 && supplyValue > maxSupplied) {
233             maxSupplied = supplyValue;
234             gTokenCollateral = markets[i];
235           }
236         }
237       }
```

Listing 3.4: `Liquidation::_getTargetMarkets()`

Recommendation Properly take the underlying token decimal into consideration to calculate the borrow/supply values of the borrower.

Status This issue has been fixed in the following commit: 717267a0.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `LineaBank` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., withdraw raising token/offering token from `preSale`). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `SaleLabOverflowFarm` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the `owner` account is privileged to set the white list who can claim more offering tokens for each raising token than normal users, withdraw any amount of the raising/offering tokens from the contract which may contain the refund to users or the offering tokens that should be harvested to users, etc.

```
    function setWhitelist(address _addr, bool isWhiteUser) external onlyOwner {
        whitelist[_addr] = isWhiteUser;
    }

    function setWhitelists(address[] calldata _addrs, bool isWhiteUser) external
        onlyOwner {
        for (uint256 i = 0; i < _addrs.length; i++) {
            whitelist[_addrs[i]] = isWhiteUser;
        }
    }

    function withdrawRaisingToken(uint256 _amount) external onlyOwner {
        require(block.timestamp > endTime, "not withdraw time");
        require(_amount <= address(this).balance, "not enough token");
        _safeTransferETH(msg.sender, _amount);
    }

    function withdrawOfferingToken(uint256 _amount) external onlyOwner {
        require(block.timestamp > endTime, "not withdraw time");
        require(_amount <= offeringToken.balanceOf(address(this)), "not enough token");
        address(offeringToken).safeTransfer(msg.sender, _amount);
    }
```

Listing 3.5: Example Privileged Operations in `SaleLabOverflowFarm`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the 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 administrative privileges to the intended `DAO`-like governance contract. And 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 confirmed the `owner` will be transferred to a multi-sig account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `LineaBank` protocol, which is a simple yet outstanding lending protocol built on the `Linea` (`ConsenSys zkEVM`). `LineaBank` gives users full control over their funds and offers competitive interest rates through a decentralized market that eliminates intermediaries. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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