SMART CONTRACT AUDIT REPORT

for

LayerBank

Prepared By: Xiaomi Huang

PeckShield

October 6, 2023

Document Properties

Client	LayerBank
Title	Smart Contract Audit Report
Target	LayerBank
Version	1.0
Author	Xuxian Jiang
Auditors	Jonathan Zhao, Colin Zhong, Jianzhuo Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 6, 2023	Xuxian Jiang	Final Release
1.0-rc1	September 28, 2023	Xuxian Jiang	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 LayerBank 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 LayerBank

LayerBank is 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. Moreover, it enables users to utilize their cryptocurrencies by supplying collateral to the protocol that may be borrowed by pledging over-collateralized cryptocurrencies. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of LayerBank

Item	Description
Name	LayerBank
Website	https://layerbank.finance
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 6, 2023

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

- https://github.com/layerbank/lineabank.git (4ece9b1)

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

- https://github.com/layerbank/lineabank.git (17f6856)

1.2 About PeckShield

PeckShield Inc. [13] 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 \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [12]:

- 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

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

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) [11], 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 `LayerBank` 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	1	
Medium	3	
Low	6	
Informational	0	
Total	10	

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 1 high-severity vulnerability, 3 medium-severity vulnerabilities, and 6 low-severity vulnerabilities.

Table 2.1: Key LayerBank Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved _extendLock() Logic in xLAB	Business Logic	Resolved
PVE-002	Low	Improved Approve Management in Leverager	Coding Practices	Resolved
PVE-003	Medium	Improper Return Initialization in RewardController	Business Logic	Resolved
PVE-004	Low	Revisited utilizationRate() Logic in RateModelSlope	Business Logic	Resolved
PVE-005	Low	Possible Precision Issue in LToken::_redeem()	Numeric Errors	Resolved
PVE-006	Low	Removal of Implicit Decimal Assumption in Market	Business Logic	Resolved
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-008	Low	Incorrect earnedBalances Calculation in RewardController	Business Logic	Resolved
PVE-009	Low	Possible Costly LToken From Improper Market Initialization	Time And State	Resolved
PVE-010	High	Possible Draining of Funds Via ExchangeRate Manipulation	Business Logic	Resolved

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 Improved _extendLock() Logic in xLAB

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: xLAB
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

In the LayerBank protocol, there is a core xLAB contract that allows users to lock protocol tokens LAB to obtain voting power. While reviewing the lock-extending logic, we notice a logic issue that needs to be addressed.

To elaborate, we show below the code snippet of the _extendLock() function. As the name indicates, it is used to extend an existing lock. By design, it requires either the new unlock time is later than previous unlock time or the new voting power is greater than the previous one. However, it comes to our attention that the adjustment of voting power blindly assumes the new voting power is always greater than the previous one. In other words, when the new unlock time is later than previous unlock time, it is possible to have a smaller new voting power, which will revert the current execution. To fix, we may need to accordingly burn the lost voting power that can be computed by deducting the new voting power from the previous one.

```
function _extendLock(address account, uint256 slot, uint256 lockDuration) private nonReentrant whenNotPaused {
  require(lockDuration >= MIN_LOCK_DURATION && lockDuration <= MAX_LOCK_DURATION, "lockDuration is out of range");

  uint256 lockCount = users[account].locks.length;
  require(slot < lockCount, "invalid slot");

  uint256 originalUnlockTime = uint256(users[account].locks[slot].unlockTime);
  uint256 lockedAmount = uint256(users[account].locks[slot].lockedAmount);
  uint256 originalVeAmount = uint256(users[account].locks[slot].veAmount);
```

```
    uint256 newUnlockTime = block.timestamp + lockDuration;
    uint256 newVeAmount = calcVeAmount(lockedAmount, lockDuration);

    require(originalUnlockTime < newUnlockTime  originalVeAmount < newVeAmount, "invalid
        lockDuration");

    users[account].locks[slot].unlockTime = uint48(newUnlockTime);
    users[account].locks[slot].veAmount = newVeAmount;

    _mint(account, newVeAmount.sub(originalVeAmount));
    _updateUserBalanceHistory(account);
    _updateLABDistributorBoostedInfo(account);

    emit ExtendLock(account, slot, newUnlockTime, lockedAmount, originalVeAmount,
        newVeAmount);
  }
```

Listing 3.1: `xLAB::_extendLock()`

Recommendation Revisit the above `_extendLock()` function to properly adjust the resulting voting power.

Status This issue has been fixed in the following commit: 17f6856.

3.2 Improved Approve Management in Leverager

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Leverager`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

The `LayerBank` protocol has a `Leverager` contract to faciliate the user's leveraged operations, which naturally involves the token approval to the given markets. Our analysis shows that current approve management can be improved.

In the following, we show the code snippet from the related `loop()` routine. As the name indicates, this routine basically loops the deposit and borrow of an asset. However, the given `lToken` argument is not validated, which may be used to approve the spending to an external untrusted contract. Specifically, if a malicious user provides a 0 `loopCount`, this routine will simply invoke `asset.safeApprove(address(lToken), uint256(-1))` (line 73) where `lToken` is directly provided by the user and should not be trusted.

```
    function loop(
        address lToken,
        uint256 amount,
        uint256 borrowRatio,
        uint256 loopCount,
        bool isBorrow
    ) external {
        require(borrowRatio <= RATIO_DIVISOR, "Invalid ratio");
        address asset = ILToken(lToken).underlying();

        // true when the loop without deposit tokens
        if (!isBorrow) {
            asset.safeTransferFrom(msg.sender, address(this), amount);
        }
        if (IBEP20(asset).allowance(address(this), address(lToken)) == 0) {
            asset.safeApprove(address(lToken), uint256(-1));
        }
        if (IBEP20(asset).allowance(address(this), address(core)) == 0) {
            asset.safeApprove(address(core), uint256(-1));
        }

        if (!isBorrow) {
            core.supplyBehalf(msg.sender, lToken, amount);
        }

        for (uint256 i = 0; i < loopCount; i++) {
            amount = amount.mul(borrowRatio).div(RATIO_DIVISOR);
            core.borrowBehalf(msg.sender, lToken, amount);
            core.supplyBehalf(msg.sender, lToken, amount);
        }
    }
```

Listing 3.2: `Leverager::loop()`

Recommendation Revise the above routine to properly validate the user input. A similar issue is also present in another routine `RewardController::setXLAB()`.

Status This issue has been fixed in the following commit: 17f6856.

3.3 Improper Return Initialization in RewardController

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `RewardController`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

To facilitate the reward distribution, the `LayerBank` protocol has a `RewardController` contract. It allows to lock the reward amount for a specified `vestDuration`. In the meantime, it also supports the early withdrawal of rewards, which may charge certain penalty if not expired. Our analysis shows the early withdrawal logic needs to be revisited.

To elaborate, we show below the code snippet of the `_ieeWithdrawableBalances()` function. This function has a rather straightforward logic in locating the respective locked funds and compute the withdrawal amount as well as the penalty amount. We notice if the given `unlockTime` does not match any existing entry, the return value of `index` is not initialized and has the default value of 0, which may be mis-interpreted as the first entry. With that, we need to initialize the return value of `index = uint256(-1)` to avoid unnecessary mis-interpretation.

```
function _ieeWithdrawableBalances(
  address user,
  uint256 unlockTime
) internal view returns (uint256 amount, uint256 penaltyAmount, uint256 burnAmount,
    uint256 index) {
  for (uint256 i = 0; i < userEarnings[user].length; i++) {
    if (userEarnings[user][i].unlockTime == unlockTime) {
      (amount, , penaltyAmount, burnAmount) = _penaltyInfo(userEarnings[user][i]);
      index = i;
      break;
    }
  }
}
```

Listing 3.3: `RewardController::_ieeWithdrawableBalances()`

Recommendation Revisit the above `_ieeWithdrawableBalances()` function to properly compute the locked entry for withdrawal.

Status This issue has been fixed in the following commit: 17f6856.

3.4 Revisited utilizationRate() Logic in RateModelSlope

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RateModelSlope`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

As a lending protocol, the `LayerBank` protocol has a `RateModelSlope` contract to compute latest borrow/supply rates. Note both rates depend on the utilization rates of given market. Our analysis shows that current utilization rate calculation needs to be improved.

To elaborate, we show below the implementation of the related `utilizationRate()` function. It has a basic logic in computing the utilization rate as `borrow / (cash + borrow - reserve)` (line 45). However, in the corner case when `reserve > cash + borrow`, the intended utilization rate should be `1e18`, not current `0` (line 44).

```
43 function utilizationRate(uint256 cash, uint256 borrows, uint256 reserves) public pure
       returns (uint256) {
44   if (reserves >= cash.add(borrows)) return 0;
45   return Math.min(borrows.mul(1e18).div(cash.add(borrows).sub(reserves)), 1e18);
46 }
```

Listing 3.4: `RateModelSlope::_extendLock()`

Recommendation Revisit the above routine to compute the right utilization rate when all available funds are already borrowed.

Status This issue has been fixed in the following commit: 17f6856.

3.5 Possible Precision Issue in LToken::_redeem()

- ID: PVE-005
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `LToken`
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [1]

Description

The `LayerBank` protocol is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying and borrowing users, i.e., `mint()/redeem()` and `borrow()/repay()`. While reviewing the redeem logic, we notice the current implementation has a precision issue that has been reflected in a recent `HundredFinance` hack.

To elaborate, we show below the related `_redeem()` routine. As the name indicates, this routine is designed to redeems `LToken` in exchange for the underlying asset. When the user indicates the underlying asset amount (via `redeemUnderlying()`), the respective `lAmountToRedeem` is computed as `uAmountIn.mul(1e18).div(exchangeRate())` (line 248). Unfortunately, the current approach may unintentionally introduce a precision issue by computing the `lAmountToRedeem` amount against the protocol. Specifically, the resulting flooring-based division introduces a precision loss, which may be just a small number but plays a critical role when certain boundary conditions are met – as demonstrated in the recent `HundredFinance` hack: https://blog.hundred.finance/15-04-23-hundred-finance-hack-post-mortem-d895b618cf33.

```
242  function _redeem(address account, uint256 lAmountIn, uint256 uAmountIn) private returns (uint256) {
243    require(lAmountIn == 0  uAmountIn == 0, "LToken: one of lAmountIn or uAmountIn must be zero");
244    require(totalSupply >= lAmountIn, "LToken: not enough total supply");
245    require(getCash() >= uAmountIn  uAmountIn == 0, "LToken: not enough underlying");
246    require(getCash() >= lAmountIn.mul(exchangeRate()).div(1e18)  lAmountIn == 0, "LToken: not enough underlying");

248    uint lAmountToRedeem = lAmountIn > 0 ? lAmountIn : uAmountIn.mul(1e18).div(exchangeRate());
249    uint uAmountToRedeem = lAmountIn > 0 ? lAmountIn.mul(exchangeRate()).div(1e18) : uAmountIn;

251    require(
252      IValidator(core.validator()).redeemAllowed(address(this), account, lAmountToRedeem),
253      "LToken: cannot redeem"
254    );

256    updateSupplyInfo(account, 0, lAmountToRedeem);
257    _doTransferOut(account, uAmountToRedeem);

259    emit Transfer(account, address(0), lAmountToRedeem);
260    emit Redeem(account, uAmountToRedeem, lAmountToRedeem);
261    return uAmountToRedeem;
262  }
```

Listing 3.5: `LToken::_redeem()`

Recommendation Properly revise the above routine to ensure the precision loss needs to be computed in favor of the protocol, instead of the user. In particular, we need to ensure that markets are never empty by minting small `LToken` balances at the time of market creation so that we can prevent the rounding error being used maliciously. A deposit as small as 1 wei is sufficient.

Status The issue has been resolved by the team to ensure the market will never be empty.

3.6 Removal of Implicit Decimal Assumption in Market

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Market`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

The `LayerBank` protocol supports a number of `markets`. While reviewing the core `Market` contract, we notice an implicit assumption, i.e., the underlying asset has its decimals no larger than 18. Note that though most existing tokens satisfy this implicit assumption, we suggest to enforce it when the market is being initialized to remove this implicit assumption.

To elaborate, we show below the related `setUnderlying()` routine, which is used to initialize the underlying asset. We can add the following statement, i.e., `require( IBEP20(_underlying).decimals ()<= 18)`, to remove this assumption.

```
function setUnderlying(address _underlying) public onlyOwner {
  require(_underlying != address(0), "GMarket: invalid underlying address");
  require(underlying == address(0), "GMarket: set underlying already");
  underlying = _underlying;
}
```

Listing 3.6: `Market::setUnderlying()`

Recommendation Revisit the above routine to properly enforce the supported market has its decimals no larger than 18.

Status This issue has been resolved by ensuring all supported assets meet the above assumption.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

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

```
function setLABDistributor(address _labDistributor) external onlyKeeper {
  require(_labDistributor != address(0), "Core: invalid labDistributor address");
  labDistributor = ILABDistributor(_labDistributor);
  emit LABDistributorUpdated(_labDistributor);
}

function setRebateDistributor(address _rebateDistributor) external onlyKeeper {
  require(_rebateDistributor != address(0), "Core: invalid rebateDistributor address");
  rebateDistributor = _rebateDistributor;
  emit RebateDistributorUpdated(_rebateDistributor);
}

function setLeverager(address _leverager) external onlyKeeper {
  require(_leverager != address(0), "Core: invalid leverager address");
  leverager = _leverager;
  emit LeveragerUpdated(_leverager);
}

/// @notice close factor
/// @dev keeper address
/// @param newCloseFactor
function setCloseFactor(uint256 newCloseFactor) external onlyKeeper {
  require(
    newCloseFactor >= Constant.CLOSE_FACTOR_MIN && newCloseFactor <= Constant.CLOSE_FACTOR_MAX,
    "Core: invalid close factor"
  );
  closeFactor = newCloseFactor;
  emit CloseFactorUpdated(newCloseFactor);
}
```

```
function setCollateralFactor(
  address lToken,
  uint256 newCollateralFactor
) external onlyKeeper onlyListedMarket(lToken) {
  require(newCollateralFactor <= Constant.COLLATERAL_FACTOR_MAX, "Core: invalid
      collateral factor");
  if (newCollateralFactor != 0 && priceCalculator.getUnderlyingPrice(lToken) == 0) {
    revert("Core: invalid underlying price");
  }

  marketInfos[lToken].collateralFactor = newCollateralFactor;
  emit CollateralFactorUpdated(lToken, newCollateralFactor);
}
```

Listing 3.7: Example Privileged Operations in `CoreAdmin`

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.

3.8 Incorrect earnedBalances Calculation in RewardController

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardController`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.3, the `LayerBank` protocol has a `RewardController` contract to facilitate the reward distribution. This contract provides a helper routine to query the earned balances for a given user. Our analysis shows this helper routine needs to be improved.

To elaborate, we show below the related code snippets of the `earnedBalances()` routine. It basically iterates each locked item and computes the penalty amount (or the unlocked amount if already unlocked). The penalty amount is calculated via another helper `_ieeWithdrawableBalances()`, which has a number of return arguments. It comes to our attention that the penalty amount should be computed by calling `_penaltyInfo(userEarnings[user][i])` and the penalty amount is in the third return value, not the second (line 120).

```
109  function earnedBalances(
110    address user
111  ) public view override returns (uint256 total, uint256 unlocked, EarnedBalance[]
       memory earningsData) {
112    unlocked = balances[user].unlocked;
113    LockedBalance[] storage earnings = userEarnings[user];
114    uint256 idx;
115    for (uint256 i = 0; i < earnings.length; i++) {
116      if (earnings[i].unlockTime > block.timestamp) {
117        if (idx == 0) {
118          earningsData = new EarnedBalance[](earnings.length - i);
119        }
120        (, uint256 penaltyAmount, , ) = _ieeWithdrawableBalances(user, earnings[i].
             unlockTime);
121        earningsData[idx].amount = earnings[i].amount;
122        earningsData[idx].unlockTime = earnings[i].unlockTime;
123        earningsData[idx].penalty = penaltyAmount;
124        idx++;
125        total = total.add(earnings[i].amount);
126      } else {
127        unlocked = unlocked.add(earnings[i].amount);
128      }
129    }
130    return (total, unlocked, earningsData);
131  }
```

Listing 3.8: `RebateDistributor::earnedBalances()`

Recommendation Revisit the above routine to properly compute the penalty amount if the rewarded item is locked.

Status This issue has been fixed in the following commit: 17f6856.

3.9 Possible Costly LToken From Improper Market Initialization

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LToken`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [3]

Description

The `LayerBank` protocol allows users to deposit supported assets and get in return the share to represent the market pool ownership. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `supply()` routine, which is used for participating users to deposit the supported assets and get respective pool shares in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
90  function supply(address account, uint256 uAmount) external payable override accrue
        onlyCore returns (uint256) {
91    uint256 exchangeRate = exchangeRate();
92    uAmount = underlying == address(ETH) ? msg.value : uAmount;
93    uAmount = _doTransferIn(account, uAmount);
94    uint256 lAmount = uAmount.mul(1e18).div(exchangeRate);
95    require(lAmount > 0, "LToken: invalid lAmount");
96    updateSupplyInfo(account, lAmount, 0);

98    emit Mint(account, lAmount);
99    emit Transfer(address(0), account, lAmount);
100   return lAmount;
101 }
```

Listing 3.9: `LToken::supply()`

```
function exchangeRate() public view override returns (uint256) {
  if (totalSupply == 0) return 1e18;
  Constant.AccrueSnapshot memory snapshot = pendingAccrueSnapshot();
  return getCashPrior().add(snapshot.totalBorrow).sub(snapshot.totalReserve).mul(1e18)
      .div(totalSupply);
}
```

Listing 3.10: `Market::exchangeRate()`

Specifically, when the pool is being initialized (line 91), the share value directly takes the value of `lAmount = uAmount.mul(1e18).div(exchangeRate)` (line 94), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `lAmount = uAmount = 1 WEI`. With that, the actor can further deposit a huge amount of the underlying assets with the goal of making the pool share extremely expensive.

An extremely expensive pool share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current deposit logic to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

Status The issue has been resolved as the team plans to follow a guarded launch so that a trusted user will be the first to deposit.

3.10 Possible Draining of Funds Via ExchangeRate Manipulation

- ID: PVE-010
- Severity: High
- Likelihood: High
- Impact: High
- Target: `LToken`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

As mentioned earlier, the `LayerBank` protocol is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities. While reviewing the exchange rate calculation, we notice the current implementation resets the exchange rate to `1e18` if the `totalSupply` is smaller than the configured `DUST` threshold. This design may need to be revisited.

To elaborate, we show below the related `exchangeRate()/updateSupplyInfo()` routines. The first routine is used to compute current exchange rate while the second routine updates the user supply information as well as the total supply. It comes to our attention that the exchange rate will be reset to `1e18`, which allows a malicious actor to manipulate and steal the market funds.[1]

```
function exchangeRate() public view override returns (uint256) {
  if (totalSupply == 0) return 1e18;
  Constant.AccrueSnapshot memory snapshot = pendingAccrueSnapshot();
  return getCashPrior().add(snapshot.totalBorrow).sub(snapshot.totalReserve).mul(1e18)
      .div(totalSupply);
}
```

Listing 3.11: `Market::exchangeRate()`

```
function updateSupplyInfo(address account, uint256 addAmount, uint256 subAmount)
    internal {
  accountBalances[account] = accountBalances[account].add(addAmount).sub(subAmount);
  totalSupply = totalSupply.add(addAmount).sub(subAmount);

  totalSupply = (totalSupply < DUST) ? 0 : totalSupply;
}
```

Listing 3.12: `Market::updateSupplyInfo()`

Recommendation Revisit the above routine to properly ensure the exchange rate will be not abused to steal market funds.

Status This issue has been resolved by following the above suggestion.

[1] A delicate scenario has been prepared and shared separately to the protocol team.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `LayerBank` protocol, which is 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. Moreover, it enables users to utilize their cryptocurrencies by supplying collateral to the protocol that may be borrowed by pledging over-collateralized cryptocurrencies. 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

[7] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

[10] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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