SMART CONTRACT AUDIT REPORT

for

ZeroLend

Prepared By: Xiaomi Huang

PeckShield

February 3, 2024

Document Properties

Client	ZeroLend
Title	Smart Contract Audit Report
Target	ZeroLend
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	January 15, 2024	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 source code of the `ZeroLend` 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 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 ZeroLend

`ZeroLend` is one of the largest lending protocols on `zkSync` that allows users to borrow/lend native crypto assets efficiently. `ZeroLend`'s core product is its decentralized non-custodial liquidity market. It is a fork of `Prisma` with changes in its own incentive mechanisms. `ZeroLend` also has a yield-bearing stablecoin (`ONEZ`) that accrues interest over time from it's core lending market and uses `LayerZero` to enable cross-chain lending. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The ZeroLend

Item	Description
Name	ZeroLend
Website	https://zerolend.xyz/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 3, 2024

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

- https://github.com/zerolend/prisma-fork.git (3fed7cc)

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

- https://github.com/zerolend/prisma-fork.git (TBD)

1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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

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

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) [8], 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 implementation of the `ZeroLend` 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	2	■■
Low	5	■■■■■
Informational	1	■
Total	8	

We have so far identified a list of potential issues. 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 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 1 informational suggestion.

Table 2.1: Key ZeroLend Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Flushing Logic in WETHDelegate	Business Logic	
PVE-002	Medium	Inaccurate Reward Calculation Logic in StabilityPool	Business Logic	
PVE-003	Low	Improved Collateral Gain Calculation Logic in StabilityPool	Business Logic	
PVE-004	Low	Improved Liquidation Logic in LiquidationManager	Business Logic	
PVE-005	Low	Improved Validation in PrismaToken::permit()	Coding Practices	
PVE-006	Informational	Revisited Proposal Vote Weight in AdminVoting	Business Logic	
PVE-007	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	
PVE-008	Medium	Trust Issue of Admin Keys	Security Features	

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Improved Flushing Logic in WETHDelegate

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: WETHDelegate
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To facilitate user interactions, the ZeroLend protocol has a helper WETHDelegate constract so that users can conveniently make use of native coins (e.g., Ether). While examining the related user interactions, we notice an underlying helper routine can be improved.

In the following, we shows the implementation of the related _flush() helper routine. It has a rather straightforward logic in withdrawing from the lending protocol and then withdrawing from WETH back to the calling user. However, the WETH withdrawal is better performed as weth.withdraw(weth.balanceOf(this)), instead of weth.withdraw(balColl) (line 42).

```
35     function _flush(address to) internal override {
36         uint256 balColl = collateral.balanceOf(address(this));
37         uint256 balDebt = debt.balanceOf(address(this));

39         // withdraw from the lending protocol and withdraw from weth
40         if (balColl > 0) {
41             collateral.burnTo(address(this), balColl);
42             weth.withdraw(balColl);

44             // we use address(this).balance because the difference from balColl and
45             // address(this).balance is accumulated the yield
46             (bool callSuccess, ) = to.call{value: address(this).balance}("");
47             require(callSuccess, "eth transfer failed");
48         }

50         if (balDebt > 0) debt.transfer(to, balDebt);
```

```
    }
```

Listing 3.1: `WETHDelegate::_flush()`

Recommendation Improve the above-mentioned helper routine in fully withdrawing the `WETH` balance.

Status

3.2 Inaccurate Reward Calculation Logic in StabilityPool

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

Description

The `ZeroLend` protocol has a core `StabilityPool` contract that is the first line of defense in maintaining entire protocol solvency. It achieves that by acting as the source of liquidity to repay debt from liquidated borrows. And the supplying users into `StabilityPool` will be rewarded with the accumulated collateral and extra protocol tokens. However, our analysis shows the related reward calculation may be improved.

To elaborate, we show below the implementation of the related reward routine, i.e., `claimableReward()`. It basically computes the gain earned by a deposit since its last snapshots were taken. Specifically, the gain is computed according to the following formula: `d0 * (G - G(0))/P(0)`, where `d0` is the last recorded deposit value and `G(0)` and `P(0)` are the depositor's snapshots of the sum `G` and product `P`, respectively.

```
    function claimableReward(
        address _depositor
    ) external view returns (uint256) {
        uint256 totalDebt = totalDebtTokenDeposits;
        uint256 initialDeposit = accountDeposits[_depositor].amount;

        if (totalDebt == 0  initialDeposit == 0) {
            return storedPendingReward[_depositor];
        }
        uint256 prismaNumerator = (_vestedEmissions() * DECIMAL_PRECISION) +
            lastPrismaError;
        uint256 prismaPerUnitStaked = prismaNumerator / totalDebt;
        uint256 marginalPrismaGain = prismaPerUnitStaked * P;

```

```
700         Snapshots memory snapshots = depositSnapshots[_depositor];
701         uint128 epochSnapshot = snapshots.epoch;
702         uint128 scaleSnapshot = snapshots.scale;
703         uint256 firstPortion;
704         uint256 secondPortion;
705         if (scaleSnapshot == currentScale) {
706             firstPortion =
707                 epochToScaleToG[epochSnapshot][scaleSnapshot] -
708                 snapshots.G +
709                 marginalPrismaGain;
710             secondPortion =
711                 epochToScaleToG[epochSnapshot][scaleSnapshot + 1] /
712                 SCALE_FACTOR;
713         } else {
714             firstPortion =
715                 epochToScaleToG[epochSnapshot][scaleSnapshot] -
716                 snapshots.G;
717             secondPortion =
718                 (epochToScaleToG[epochSnapshot][scaleSnapshot + 1] +
719                     marginalPrismaGain) /
720                 SCALE_FACTOR;
721         }
722 
723         return
724             storedPendingReward[_depositor] +
725             (initialDeposit * (firstPortion + secondPortion)) /
726             snapshots.P /
727             DECIMAL_PRECISION;
728     }
```

Listing 3.2: StabilityPool ::claimableReward()

It comes to our attention that `marginalPrismaGain` is largely dependent on `scaleSnapshot` and `currentScale`. In particular, when `scaleSnapshot > currentScale + 1`, there is a need to reset `marginalPrismaGain = 0`, which is not the case yet in current logic (lines 713-721).

Recommendation Revise the above `claimableReward()` routine to properly compute the claimable rewards.

Status

3.3 Improved Collateral Gain Calculation Logic in StabilityPool

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StabilityPool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, `StabilityPool` is the first line of defense in maintaining entire protocol solvency. While acting as the source of liquidity to repay debt from liquidated positions, the supplying users are rewarded with the accumulated collateral and extra protocol tokens. In the process of examining the collateral-accumulating logic, we notice its implementation may be improved.

To elaborate, we show below the related `_accrueDepositorCollateralGain()` routine. As the name indicates, it is used to accumulate the collateral gain for supplying users. We notice this routine returns a boolean indicating whether there is an actual gain for the given user. However, the return value should be indicated if there is the non-zero increased gain in the calculation of `depositorGains[i]` (line 671), instead of always `true` if the execution logic enters the `for`-loop (line 666).

```
641     function _accrueDepositorCollateralGain(
642         address _depositor
643     ) private returns (bool hasGains) {
644         uint80[256] storage depositorGains = collateralGainsByDepositor[
645             _depositor
646         ];
647         uint256 collaterals = collateralTokens.length;
648         uint256 initialDeposit = accountDeposits[_depositor].amount;
649         hasGains = false;
650         if (initialDeposit == 0) {
651             return hasGains;
652         }
653
654         uint128 epochSnapshot = depositSnapshots[_depositor].epoch;
655         uint128 scaleSnapshot = depositSnapshots[_depositor].scale;
656         uint256 P_Snapshot = depositSnapshots[_depositor].P;
657
658         uint256[256] storage sums = epochToScaleToSums[epochSnapshot][
659             scaleSnapshot
660         ];
661         uint256[256] storage nextSums = epochToScaleToSums[epochSnapshot][
662             scaleSnapshot + 1
663         ];
664         uint256[256] storage depSums = depositSums[_depositor];
665
666         for (uint256 i = 0; i < collaterals; i++) {
```

```
            if (sums[i] == 0) continue; // Collateral was overwritten or not gains
            hasGains = true;
            uint256 firstPortion = sums[i] - depSums[i];
            uint256 secondPortion = nextSums[i] / SCALE_FACTOR;
            depositorGains[i] += uint80(
                (initialDeposit * (firstPortion + secondPortion)) /
                    P_Snapshot /
                    DECIMAL_PRECISION
            );
        }
        return (hasGains);
    }
```

Listing 3.3: StabilityPool :: _accrueDepositorCollateralGain()

Recommendation Improve the given `_accrueDepositorCollateralGain()` routine to accurately reflect whether the user has any actual collateral gain increase.

Status

3.4 Improved Liquidation Logic in LiquidationManager

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LiquidationManager`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In `ZeroLend`, there is a built-in liquidation contract `LiquidationManager` that handles liquidations for every active collateral within the protocol. While examining current batch-liquidation logic, we notice the implementation can be improved.

In the following, we show the related code snippet from the affected function `batchLiquidateTroves()`. We notice there is a local variable `troveCount` to keep track of the total `trove` count. However, the `trove` count is not decreased (lines 383, 388, and 396) while iterating the list of current `troves`. `reward = pendingBaseReward(tokenID)+ vBaseClaimed[tokenID]`.

```
351         if (troveIter < length && troveCount > 1) {
352             // second iteration round, if we receive a trove with ICR > MCR and need to
                    track TCR
353             (
354                 uint256 entireSystemColl,
355                 uint256 entireSystemDebt
356             ) = borrowerOperations.getGlobalSystemBalances();
```

```
            entireSystemColl -=
                totals.totalCollToSendToSP *
                troveManagerValues.price;
            entireSystemDebt -= totals.totalDebtToOffset;
            while (troveIter < length && troveCount > 1) {
                address account = _troveArray[troveIter];
                uint ICR = troveManager.getCurrentICR(
                    account,
                    troveManagerValues.price
                );
                unchecked {
                    ++troveIter;
                }
                if (ICR <= _100pct) {
                    singleLiquidation = _liquidateWithoutSP(
                        troveManager,
                        account
                    );
                } else if (ICR < troveManagerValues.MCR) {
                    singleLiquidation = _liquidateNormalMode(
                        troveManager,
                        account,
                        debtInStabPool,
                        troveManagerValues.sunsetting
                    );
                } else {
                    if (troveManagerValues.sunsetting) continue;
                    uint256 TCR = PrismaMath._computeCR(
                        entireSystemColl,
                        entireSystemDebt
                    );
                    if (TCR >= CCR  ICR >= TCR) continue;
                    singleLiquidation = _tryLiquidateWithCap(
                        troveManager,
                        account,
                        debtInStabPool,
                        troveManagerValues.MCR,
                        troveManagerValues.price
                    );
                    if (singleLiquidation.debtToOffset == 0) continue;
                }

                debtInStabPool -= singleLiquidation.debtToOffset;
                entireSystemColl -=
                    (singleLiquidation.collToSendToSP +
                        singleLiquidation.collSurplus) *
                    troveManagerValues.price;
                entireSystemDebt -= singleLiquidation.debtToOffset;
                _applyLiquidationValuesToTotals(totals, singleLiquidation);
                unchecked {
                    --troveCount;
                }
```

```
            }
        }
```

Listing 3.4: LiquidationManager::batchLiquidateTroves()

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

Recommendation Revise the above-mentioned routines to properly maintain the internal variable troveCount.

Status

3.5 Improved Validation in PrismaToken::permit()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PrismaToken
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The ZeroLend protocol has a token contract PrismaToken that supports the EIP2612 functionality. In particular, the permit() function is introduced to simplify the token transfer process.

To elaborate, we show below this helper routine from the PrismaToken contract. This routine ensures that the given owner is indeed the one who signs the approve request. Note that the internal implementation makes use of the ecrecover() precompile for validation. It comes to our attention that the precompile-based validation needs to properly ensure the signer, i.e., owner, is not equal to address(0).

```
    function permit(
        address owner,
        address spender,
        uint256 amount,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external override {
        require(deadline >= block.timestamp, "PRISMA: expired deadline");
        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                domainSeparator(),
                keccak256(
                    abi.encode(
```

```
                        permitTypeHash,
                        owner,
                        spender,
                        amount,
                        _nonces[owner]++,
                        deadline
                    )
                )
            )
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress == owner, "PRISMA: invalid signature");
        _approve(owner, spender, amount);
    }
```

Listing 3.5: `PrismaToken::permit()`

Recommendation Strengthen the `permit()` routine to ensure the `owner` is not equal to `address(0)`.

Status

3.6 Revisited Proposal Vote Weight in AdminVoting

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AdminVoting`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In `ZeroLend`, there is a built-in `AdminVoting` contract to vote proposals on behalf of `DAO Admin`. It is authorized to execute arbitrary function calls after a required percentage of `PRISMA` lockers have signaled in favor of performing the action. While examining the actual proposal-voting logic, we notice the design may be revisited.

In the following, we show the code snippet from the related function `voteForProposal()`. Our analysis shows that for any proposal, the `DAO Admin` always has the same maximum `accountWeight` regardless how many proposals have been voted. A better alternative would be to specify the `accountWeightAtWeek` voting power at a specific week. And for all proposals voted at that week, the total weight should not exceed the specified `accountWeightAtWeek`.

```
    function voteForProposal(
        address account,
```

```
        uint256 id,
        uint256 weight
    ) external callerOrDelegated(account) {
        require(id < proposalData.length, "Invalid ID");
        require(accountVoteWeights[account][id] == 0, "Already voted");

        Proposal memory proposal = proposalData[id];
        require(!proposal.processed, "Proposal already processed");
        require(
            proposal.createdAt + VOTING_PERIOD > block.timestamp,
            "Voting period has closed"
        );

        uint256 accountWeight = tokenLocker.getAccountWeightAt(
            account,
            proposal.week
        );
        if (weight == 0) {
            weight = accountWeight;
            require(weight > 0, "No vote weight");
        } else {
            require(weight <= accountWeight, "Weight exceeds account weight");
        }

        accountVoteWeights[account][id] = weight;
        uint40 updatedWeight = uint40(proposal.currentWeight + weight);
        proposalData[id].currentWeight = updatedWeight;
        bool hasPassed = updatedWeight >= proposal.requiredWeight;

        if (proposal.canExecuteAfter == 0 && hasPassed) {
            uint256 canExecuteAfter = block.timestamp + MIN_TIME_TO_EXECUTION;
            proposalData[id].canExecuteAfter = uint32(canExecuteAfter);
            emit ProposalHasMetQuorum(id, canExecuteAfter);
        }

        emit VoteCast(account, id, weight, updatedWeight, hasPassed);
    }
```

Listing 3.6: `AdminVoting::voteForProposal()`

Recommendation Revisit the above proposal-voting routine to ensure the total voting weight of `DAO Admin` at a specific week should not exceed `accountWeightAtWeek`.

Status

3.7 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

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 3.7: 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.

In current implementation, if we examine the `CurveProxy::claimFees()` routine that is designed to claim the fee from `feeDistributor`. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 192).

```
188     function claimFees() external returns (uint256) {
```

```
        feeDistributor.claim();
        uint256 amount = feeToken.balanceOf(address(this));

        feeToken.transfer(PRISMA_CORE.feeReceiver(), amount);

        return amount;
    }
```

Listing 3.8: `CurveProxy::claimFees()`

In the meantime, we also suggest to use the safe-version of `transferFrom()` and `approve()` in other related contracts, including `WrappedLendingCollateral`, `BaseDelegate`, and `ERC20Delegate`.

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

Status

3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In `ZeroLend`, there is a privileged administrative `owner` account. The administrative account plays a critical role in governing and regulating the protocol-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `Vault` contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
    function setReceiverIsActive(
        uint256 id,
        bool isActive
    ) external onlyOwner returns (bool) {
        Receiver memory receiver = idToReceiver[id];
        require(receiver.account != address(0), "ID not set");
        receiver.isActive = isActive;
        idToReceiver[id] = receiver;
        emit ReceiverIsActiveStatusModified(id, isActive);

        return true;
    }

```

```
    /**
        @notice Set the ‘emissionSchedule‘ contract
        @dev Callable only by the owner (the DAO admin voter, to change the emission
            schedule).
             The new schedule is applied from the start of the next epoch.
     */
    function setEmissionSchedule(
        IEmissionSchedule _emissionSchedule
    ) external onlyOwner returns (bool) {
        _allocateTotalWeekly(emissionSchedule, getWeek());
        emissionSchedule = _emissionSchedule;
        emit EmissionScheduleSet(address(_emissionSchedule));

        return true;
    }

    function setBoostCalculator(
        IBoostCalculator _boostCalculator
    ) external onlyOwner returns (bool) {
        boostCalculator = _boostCalculator;
        emit BoostCalculatorSet(address(_boostCalculator));

        return true;
    }

    /**
        @notice Transfer tokens out of the vault
     */
    function transferTokens(
        IERC20 token,
        address receiver,
        uint256 amount
    ) external onlyOwner returns (bool) {
        if (address(token) == address(prismaToken)) {
            require(receiver != address(this), "Self transfer denied");
            uint256 unallocated = unallocatedTotal - amount;
            unallocatedTotal = uint128(unallocated);
            emit UnallocatedSupplyReduced(amount, unallocated);
        }
        token.safeTransfer(receiver, amount);

        return true;
    }
```

Listing 3.9: Example Privileged Operations in `Vault`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the administrative account may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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

4 Conclusion

In this audit, we have analyzed the design and implementation of the `ZeroLend` protocol, which is one of the largest lending protocols on `zkSync` that allows users to borrow/lend native crypto assets efficiently. `ZeroLend`'s core product is its decentralized non-custodial liquidity market. It is a fork of `Prisma` with changes in its own incentive mechanisms. `ZeroLend` also has a yield-bearing stablecoin (`ONEZ`) that accrues interest over time from it's core lending market and uses `LayerZero` to enable cross-chain lending. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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