



SMART CONTRACT AUDIT REPORT

for

LEND by TEN Finance



Prepared By: Patrick Lou

PeckShield
April 16, 2022

Document Properties

Client	TenFinance
Title	Smart Contract Audit Report
Target	LEND by TEN Finance
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 16, 2022	Jing Wang	Final Release
1.0-rc	March 24, 2022	Jing Wang	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About LEND by TEN Finance	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Improper Authorization In changeReduceReserveCaller()	11
3.2	Improper Handling of SupplyList in seizeInternal()	12
3.3	Sandwiched register() And transferLEND() For Airdrop	13
3.4	Uninitialized State Index DoS From Reward Activation	15
3.5	Non ERC20-Compliance Of TToken	19
3.6	Suggested Adherence Of Checks-Effects-Interactions Pattern	22
3.7	Possible Sandwich/MEV Attacks For Reduced Returns	24
3.8	Trust Issue of Admin Keys	25
4	Conclusion	27
	References	28

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the LEND by TEN Finance 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 LEND by TEN Finance

LEND by TEN Finance is a lending and borrowing protocol with the goal of developing a shared revenue fees money market. The protocol design is architected and inspired based on Compound. LEND by TEN Finance will establish pools of algorithmically derived interest rate model, based on current supply and demand of each respective asset. Suppliers and Borrowers of assets interact directly with the protocol in earning and paying a floating interest rate. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The LEND by TEN Finance Protocol

Item	Description
Name	TenFinance
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 16, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that LEND by TEN Finance assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- <https://github.com/sprapidinnovation/TenLend.git> (26042ad)

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

- <https://github.com/sprapidinnovation/TenLend.git> (ca18558)

1.2 About PeckShield

PeckShield Inc. [15] 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 [14]:

- 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.

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

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [13], 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 LEND by TEN Finance 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	1	
High	1	
Medium	4	
Low	2	
Informational	0	
Total	8	

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 critical-severity vulnerability, 1 high-severity vulnerability, 4 medium-severity vulnerability, and 2 low-severity vulnerabilities.

Table 2.1: Key LEND by TEN Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Improper Authorization In <code>changeReduceReserveCaller()</code>	Business Logic	Fixed
PVE-002	Medium	Improper Handling of <code>SupplyList</code> in <code>seizeInternal()</code>	Business Logic	Fixed
PVE-003	Medium	Sandwiched <code>register()</code> And <code>transferLEND()</code> For Airdrop	Business Logic	Confirmed
PVE-004	High	Uninitialized State Index DoS From Reward Activation	Business Logic	Fixed
PVE-005	Low	Non ERC20-Compliance Of <code>TToken</code>	Coding Practices	Partially Fixed
PVE-006	Medium	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Partially Fixed
PVE-007	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time And State	Confirmed
PVE-008	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Improper Authorization In `changeReduceReserveCaller()`

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: TToken
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

As mentioned earlier, the LEND by TEN Finance protocol is heavily forked from Compound and shares the same architectural design. The protocol extends the `cToken` contract to support sharing revenue fees of the platform with its token holders. While examining this part of logic, we notice an issue in current implementation.

```

1336     function changeReduceReserveCaller(address newCaller) external {
1337         require(msg.sender != admin);
1338         reduceReserveCaller = newCaller;
1339     }

```

Listing 3.1: TToken::changeReduceReserveCaller()

```

2667     function _reduceReservesFresh(uint reduceAmount) internal returns (uint) {
2668         // totalReserves - reduceAmount
2669         uint totalReservesNew;

2671         // Check caller is reduceReserveCaller
2672         if (msg.sender != reduceReserveCaller) {
2673             return fail(Error.UNAUTHORIZED, FailureInfo.REDUCE_RESERVES_ADMIN_CHECK);
2674         }

2676         ...

2678         // doTransferOut reverts if anything goes wrong, since we can't be sure if side
            effects occurred.

```

```

2680         (MathError mathErr, uint halfReduceAmount) = divUInt(reduceAmount, 2); //
2682         if (mathErr != MathError.NO_ERROR) {
2683             return fail(Error.MATH_ERROR, FailureInfo.REDUCE_RESERVES_VALIDATION);
2684         }
2686         doTransferOut(admin, halfReduceAmount);
2688         doTransferOut(msg.sender, halfReduceAmount);
2690         emit ReservesReduced(admin, reduceAmount, totalReservesNew);
2692         return uint(Error.NO_ERROR);
2693     }

```

Listing 3.2: TToken::_reduceReservesFresh()

To elaborate, we show above the `changeReduceReserveCaller()` and `_reduceReservesFresh()` routines. We notice the `_reduceReservesFresh()` routine is used to transfer reserves to admin by the privileged account `reduceReserveCaller`. However, in the `changeReduceReserveCaller()` routine, the sanity check is requiring `msg.sender != admin` when calling this routine, which is giving privilege control to anyone who is NOT the admin!

Recommendation Correct the above routine to properly handle the privilege control in the `changeReduceReserveCaller()` routine.

Status The issue has been fixed by this commit: 08bfab8.

3.2 Improper Handling of SupplyList in seizeInternal()

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: TToken
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

As mentioned in Section 3.1, the LEND by TEN Finance protocol is heavily forked from Compound and shares the same architectural design, including the liquidation functionality. To support viewing the supplier and borrower list, the LEND by TEN Finance protocol extends the `cToken` contract by introducing several variables, e.g., `BorrowList[]`, `SupplyList[]`, `userBorrowRecord` and `userSupplyRecord` to help reflecting user token balances changes. While examining this part of logic, we notice an issue in current implementation. To elaborate, we show below the related code snippet.

```

2399     function seizeInternal(address seizerToken, address liquidator, address borrower,
2400         uint seizeTokens) internal returns (uint) {
2401         ...
2402         (vars.mathErr, vars.borrowerTokensNew) = subUInt(accountTokens[borrower],
2403             seizeTokens);
2404         ...
2405         /* We write the previously calculated values into storage */
2406         totalReserves = vars.totalReservesNew;
2407         totalSupply = vars.totalSupplyNew;
2408         accountTokens[borrower] = vars.borrowerTokensNew;
2409         accountTokens[liquidator] = vars.liquidatorTokensNew;
2410         ...
2411         return uint(Error.NO_ERROR);
2412     }

```

Listing 3.3: TToken::seizeInternal()

Specifically, we show above the related implementation of the `seizeInternal()` routine. This routine calculates the new borrower and liquidator token balances after liquidation and writes the updated values into storage. However, our analysis shows that it only updates the balance of `TToken` and leaves borrower and supplier lists unchanged. In fact, without properly handling the modification of borrower and supplier lists, the information provided by the `BorrowList[]` and `SupplyList[]` could be misleading.

Recommendation Properly handle the borrower and supplier list modification in the `TToken::seizeInternal()` routine.

Status The issue has been fixed by this commit: 58d787a.

3.3 Sandwiched register() And transferLEND() For Airdrop

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Airdrop
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

In the `LEND` by `TEN Finance` protocol, the `Airdrop` contract allows the investor to register the airdrop and transfer the airdrop rewards every 90 days. To get the qualification for the airdrop, the protocol requires the user to have a minimum balance of `tenfi` in the holding account. While examining this part of logic, we notice an issue in current implementation. To elaborate, we show below the related routines.

```

410     function register() external nonReentrant {
411
412         require(block.timestamp < airdropStartTime,"Registration time already ended");
413
414         address user = _msgSender();
415
416         uint256 tenfi_amount = getUserTenfiBalance(user);
417
418         require(tenfi_amount >= MIN_QUALIFY_AMOUNT,"Insufficient tenfi balance to
            qualify");
419         ...
420     }

```

Listing 3.4: Airdrop::register()

```

692     function getUserTenfiBalance(address user) public view returns(uint tenfi_bal) {

693
694
695         address _tenfiAddress = TENFI_ADDRESS; //gas-savings
696         //User wallet tenfi balance
697         tenfi_bal += IERC20(_tenfiAddress).balanceOf(user);
698
699         uint256 tenfi_farm_length = registeredTenfiFarmVaultsId.length;
700         ...
701     }

```

Listing 3.5: Airdrop::getUserTenfiBalance()

```

476     function transferLEND() external nonReentrant {
477
478         /*
479         *if user status is INSIDE it means that the user is
480         * getting airdrop for 1st round
481         * getting airdrop for other rounds except 1st means previous rounds are
            successfull
482         */
483
484         address user = _msgSender();
485
486         bool flag = canWeTransferLend(user); // check user has sufficient tenfi balance
487
488         require(flag == true,"User not have sufficient tenfi or not inside the airdrop")
            ;
489         ...
490     }

```

Listing 3.6: Airdrop::transferLEND()

```

749     function canWeTransferLend(address user) public view returns(bool flag) {
750
751         if(isUserInAirdrop[user] != Status.INSIDE) {

```

```

753         flag = false;
754     }
755     else {
756
757         uint256 tenfi_amount = getUserTenfiBalance(user);
758
759         if(tenfi_amount < tenfiAtWhichUserIsRegistered[user]) {
760             flag = false;
761         }
762         else {
763             flag = true;
764         }
765     }
766 }
767 }
768 }

```

Listing 3.7: Airdrop::canWeTransferLend()

In particular, we notice in the `register()` and `canWeTransferLend()` routines, the helper routine `getUserTenfiBalance()` is checking the airdrop qualification by calculating whether the account is directly holding or staking `MIN_QUALIFY_AMOUNT` of `tenfi` tokens. However, a bad actor could flashloan `MIN_QUALIFY_AMOUNT` of `tenfi` tokens before registration or claiming rewards from the `Airdrop` contract and then repay the loan afterwards within the same transaction.

Recommendation Take into account the `tenfi` staked time as well for airdrop qualification.

Status This issue has been confirmed by the team and the team clarifies that they will resolve in their airdrop backend only where they are going to expel the users if they do not have adequate balance at the time of airdrop starting.

3.4 Uninitialized State Index DoS From Reward Activation

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: Tentroller
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The `LEND` by `TEN Finance` protocol provides incentive mechanisms that reward the protocol users. Specifically, the reward mechanism follows the same approach as the `COMP` reward in `Compound`. Our analysis on the related `LENDt` reward in `LEND` by `TEN Finance` shows the current logic needs to be improved.

To elaborate, we show below the initial logic of `setLendSpeedInternal()` that kicks off the actual minting of protocol tokens. It comes to our attention that the initial supply-side index is configured on the conditions of `LendtSupplyState[address(tToken)].index == 0` and `LendtSupplyState[address(tToken)].block == 0` (line 4384). However, for an already listed market with a current speed of 0, the first condition is indeed met while the second condition does not! The reason is that both supply-side state and borrow-side state have the associated block information updated, which is diligently performed via other helper pairs `updateLendtSupplyIndex()/updateLendtBorrowIndex()`. As a result, the `setLendSpeedInternal()` logic does not properly set up the default supply-side index and the default borrow-side index.

```

4372     function setLendtSpeedInternal(TToken tToken, uint lendtSpeed) internal {
4373         uint currentLendtSpeed = lendtSpeeds[address(tToken)];
4374         if (currentLendtSpeed != 0) {
4375             // note that LENDt speed could be set to 0 to halt liquidity rewards for a
4376             // market
4377             Exp memory borrowIndex = Exp({mantissa: tToken.borrowIndex()});
4378             updateLendtSupplyIndex(address(tToken));
4379             updateLendtBorrowIndex(address(tToken), borrowIndex);
4380         } else if (lendtSpeed != 0) {
4381             // Add the LENDt market
4382             Market storage market = markets[address(tToken)];
4383             require(market.isListed == true, "lendt market is not listed");
4384
4385             if (lendtSupplyState[address(tToken)].index == 0 && lendtSupplyState[address(
4386                 tToken)].block == 0) {
4387                 lendtSupplyState[address(tToken)] = LendtMarketState({
4388                     index: lendtInitialIndex,
4389                     block: safe32(getBlockNumber(), "block number exceeds 32 bits")
4390                 });
4391             }
4392
4393             if (lendtBorrowState[address(tToken)].index == 0 && lendtBorrowState[address(
4394                 tToken)].block == 0) {
4395                 lendtBorrowState[address(tToken)] = LendtMarketState({
4396                     index: lendtInitialIndex,
4397                     block: safe32(getBlockNumber(), "block number exceeds 32 bits")
4398                 });
4399             }
4400         }
4401         if (currentLendtSpeed != lendtSpeed) {
4402             lendtSpeeds[address(tToken)] = lendtSpeed;
4403             emit LendtSpeedUpdated(tToken, lendtSpeed);
4404         }
4405     }

```

Listing 3.8: `Tentroller::setLendtSpeedInternal()`

```

4409     function updateLendtSupplyIndex(address tToken) internal {
4410         LendtMarketState storage supplyState = lendtSupplyState[tToken];

```



```

4411     uint supplySpeed = lendtSpeeds[tToken];
4412     uint blockNumber = getBlockNumber();
4413     uint deltaBlocks = sub_(blockNumber, uint(supplyState.block));
4414     if (deltaBlocks > 0 && supplySpeed > 0) {
4415         uint supplyTokens = TToken(tToken).totalSupply();
4416         uint lendtAccrued = mul_(deltaBlocks, supplySpeed);
4417         Double memory ratio = supplyTokens > 0 ? fraction(lendtAccrued, supplyTokens
            ) : Double({mantissa: 0});
4418         Double memory index = add_(Double({mantissa: supplyState.index}), ratio);
4419         lendtSupplyState[tToken] = LendtMarketState({
4420             index: safe224(index.mantissa, "new index exceeds 224 bits"),
4421             block: safe32(blockNumber, "block number exceeds 32 bits")
4422         });
4423     } else if (deltaBlocks > 0) {
4424         supplyState.block = safe32(blockNumber, "block number exceeds 32 bits");
4425     }
4426 }

```

Listing 3.9: Tentroller::updateLendSupplyIndex()

When the reward speed is configured, since the supply-side and borrow-side state indexes are not initialized, any normal functionality such as mint() will be immediately reverted! This revert occurs inside the distributeSupplierLendt()/distributeBorrowerLendt() functions. Using the distributeSupplierLendt() function as an example, the revert is caused from the arithmetic operation sub_(supplyIndex, supplierIndex) (line 4466). Since the supplyIndex is not properly initialized, it will be updated to a smaller number from an earlier invocation of updateLendSupplyIndex() (lines 4376-4378). However, when the distributeSupplierLendt() function is invoked, the supplierIndex is reset with LendtInitialIndex (line 4463), which unfortunately reverts the arithmetic operation sub_(supplyIndex, supplierIndex)!

```

4456     function distributeSupplierLendt(address tToken, address supplier) internal {
4457         LendtMarketState storage supplyState = lendtSupplyState[tToken];
4458         Double memory supplyIndex = Double({mantissa: supplyState.index});
4459         Double memory supplierIndex = Double({mantissa: lendtSupplierIndex[tToken][
            supplier]});
4460         lendtSupplierIndex[tToken][supplier] = supplyIndex.mantissa;

4462         if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
4463             supplierIndex.mantissa = lendtInitialIndex;
4464         }

4466         Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
4467         uint supplierTokens = TToken(tToken).balanceOf(supplier);
4468         uint supplierDelta = mul_(supplierTokens, deltaIndex);
4469         uint supplierAccrued = add_(lendtAccrued[supplier], supplierDelta);
4470         lendtAccrued[supplier] = supplierAccrued;
4471         emit DistributedSupplierLendt(TToken(tToken), supplier, supplierDelta,
            supplyIndex.mantissa);

```

4472 }

Listing 3.10: Tentroller::distributeSupplierLendt()

Recommendation Properly initialize the reward state indexes in the above affected `setLendtSpeedInternal()` function. An example revision is shown as follows:

```

4372 function setLendtSpeedInternal(TToken tToken, uint lendtSpeed) internal {
4373     uint currentLendtSpeed = lendtSpeeds[address(tToken)];
4374     if (currentLendtSpeed != 0) {
4375         // note that Lendt speed could be set to 0 to halt liquidity rewards for a
            market
4376         Exp memory borrowIndex = Exp({mantissa: tToken.borrowIndex()});
4377         updateLendtSupplyIndex(address(tToken));
4378         updateLendtBorrowIndex(address(tToken), borrowIndex);
4379     } else if (lendtSpeed != 0) {
4380         // Add the Lendt market
4381         Market storage market = markets[address(tToken)];
4382         require(market.isListed == true, "lendt market is not listed");

4384         if (lendtSupplyState[address(tToken)].index == 0) {
4385             lendtSupplyState[address(tToken)] = LendtMarketState({
4386                 index: lendtInitialIndex,
4387                 block: safe32(getBlockNumber(), "block number exceeds 32 bits")
4388             });
4389         }

4391         if (lendtBorrowState[address(tToken)].index == 0) {
4392             lendtBorrowState[address(tToken)] = LendtMarketState({
4393                 index: lendtInitialIndex,
4394                 block: safe32(getBlockNumber(), "block number exceeds 32 bits")
4395             });
4396         }
4397     }

4399     if (currentLendtSpeed != lendtSpeed) {
4400         lendtSpeeds[address(tToken)] = lendtSpeed;
4401         emit LendtSpeedUpdated(tToken, lendtSpeed);
4402     }
4403 }
```

Listing 3.11: Tentroller::setLendtSpeedInternal()

Status The issue has been fixed by this commit: [dabfbb6](#).

3.5 Non ERC20-Compliance Of TToken

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: TToken
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [3]

Description

Each asset supported by the LEND by TEN Finance protocol is integrated through a so-called `tToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `tTokens`, users can earn interest through the `tToken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `tTokens` as collateral. There are currently two types of `tTokens`: `TErc20` and `TBNB`. In the following, we examine the ERC20 compliance of these `tTokens`.

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we

examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.2: Key State-Changing Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	×
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to <i>address(0x0)</i> when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the `tToken` contract. Specifically, the current `transfer()` function simply returns the related error code if the sender does not have sufficient balance to spend. A similar issue is also present in the `transferFrom()` function that does not revert when the sender does not have the sufficient balance or the message sender does not have the enough allowance.

Moreover, the ERC20 token standard also specifies that “a token contract which creates new tokens SHOULD trigger a `Transfer` event with the `_from` address set to `0x0` when tokens are created.” [1] However, current `mint()` logic emits the `Transfer` event by specifying the `CErc20/CEther` contract itself as the `_from`. For better ERC20 compliance, it is also suggested to strictly follow the

ERC20 token standard.

In the surrounding two tables, we outline the respective list of basic [view-only](#) functions (Table 3.1) and key [state-changing](#) functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional [opt-in](#) Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on <code>transfer()/transferFrom()</code> calls	—
Rebasing	The <code>balanceOf()</code> function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

Recommendation Revise the `CToken` implementation to ensure its ERC20-compliance.

Status The issue has been partially fixed by this commit: [252393c](#).

3.6 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Time and State [11]
- CWE subcategory: CWE-663 [5]

Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [17] exploit, and the recent Uniswap/Lendf.Me hack [16].

We notice there are occasions where the checks-effects-interactions principle is violated. Using the TToken as an example, the `redeemFresh()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy. For example, the interaction with the external contract (line 1936) start before effecting the update on internal states (lines 1939 – 1940), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```

1933     function redeemFresh(address payable redeemer, uint redeemTokensIn, uint
        redeemAmountIn) internal returns (uint) {
1934         require(redeemTokensIn == 0 redeemAmountIn == 0, "one of redeemTokensIn or
            redeemAmountIn must be zero");
1935         ...
1936         doTransferOut(redeemer, vars.redeemAmount);
1937
1938         /* We write previously calculated values into storage */
1939         totalSupply = vars.totalSupplyNew;
1940         accountTokens[redeemer] = vars.accountTokensNew;
1941
1942         if(vars.redeemAmount == accountTokens[redeemer]){
1943             uint256 idx = userSupplyRecord[redeemer].idx;
1944             address temp = SupplyList[idx];
1945             SupplyList[idx] = SupplyList[SupplyList.length - 1];
1946             SupplyList[SupplyList.length - 1] = temp;
1947             SupplyList.pop();

```

```
1948         delete userSupplyRecord[redeemer];
1949     }
1950
1951     /* We emit a Transfer event, and a Redeem event */
1952     emit Transfer(redeemer, address(this), vars.redeemTokens);
1953     emit Redeem(redeemer, vars.redeemAmount, vars.redeemTokens);
1954
1955     /* We call the defense hook */
1956     tentroller.redeemVerify(address(this), redeemer, vars.redeemAmount, vars.
        redeemTokens);
1957
1958     return uint(Error.NO_ERROR);
1959 }
```

Listing 3.12: TToken::redeemFresh()

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy, it is important to take precautions to thwart possible re-entrancy. The similar issue is also present in other functions, including `borrowFresh()`, `mintFresh()` and `repayBorrowFresh()`. The adherence of the checks-effects-interactions best practice is strongly recommended. We highlight that the very same issue has been exploited in a recent Cream incident [2] and therefore deserves special attention.

From another perspective, the current mitigation in applying money-market-level reentrancy protection can be strengthened by elevating the reentrancy protection at the `Comptroller`-level. In addition, each individual function can be self-strengthened by following the checks-effects-interactions principle

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions principle and utilizing the necessary `nonReentrant` modifier to block possible re-entrancy. Also consider strengthening the reentrancy protection at the protocol-level instead of at the current money-market granularity.

Status The issue has been partially fixed by this commit: 252393c.

3.7 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: RewardShare
- Category: Time and State [12]
- CWE subcategory: CWE-682 [6]

Description

The RewardShare contract has a helper routine, i.e., `distribute()`, that is designed to convert the reserves from the money market to BUSD for Stakers. It has a rather straightforward logic in calling the `_reduceReserves()` to transfer the funds and calling `_safeSwap()` to actually perform the intended token swap.

```

1053     function _safeSwap(
1054         address _uniRouterAddress,
1055         uint256 _amountIn,
1056         uint256 _slippageFactor,
1057         address[] memory _path,
1058         address _to,
1059         uint256 _deadline
1060     ) internal virtual returns(uint256) {
1061         _approveTokenIfNeeded(_path[0], _uniRouterAddress);
1062         uint256[] memory amounts =
1063             IPancakeRouter02(_uniRouterAddress).getAmountsOut(_amountIn, _path);
1064         uint256 amountOut = amounts[amounts.length.sub(1)].mul(_slippageFactor).div
            (1000);
1065         uint256 _returned = IPancakeRouter02(_uniRouterAddress)
1066             .swapExactTokensForTokens(
1067                 _amountIn,
1068                 amountOut,
1069                 _path,
1070                 _to,
1071                 _deadline
1072             )[amounts.length.sub(1)];
1073         return _returned;
1074     }

```

Listing 3.13: RewardShare::_safeSwap()

To elaborate, we show above the `_safeSwap()` routine. We notice the actual swap operation `swapExactTokensForTokens()` essentially do not specify any valid restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller return for this round of operation.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back

of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the TWAP or time-weighted average price of UniswapV2. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense. Note the same issue also exists on the another routine in the `RewardShare` contract.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been confirmed by the team. And the team clarifies that they are going to put the path for the minimum slippage possible and going to change it if slippage exceeds to higher limits.

3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [8]
- CWE subcategory: CWE-287 [4]

Description

In the LEND by TEN Finance protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., configure system parameters). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```

740     function changePenaltyPercentgae(uint[3] memory _penalty) external onlyOwner{
741         penaltyPercentage=_penalty;
742     }
```

Listing 3.14: `MultiFeeDistribution::changePenaltyPercentgae()`

```

457     function addNewConfig(TokenConfig memory config) public onlyOwner {
458
459         require(config.baseUnit > 0, "baseUnit must be greater than zero");
460
461         address uniswapMarket = config.uniswapMarket;
462         if (config.priceSource == PriceSource.REPORTER) {
```

```
463         require(uniswapMarket != address(0), "reported prices must have an anchor");
464         require(config.reporter != address(0), "reported price must have a reporter"
465             );
466         bytes32 symbolHash = config.symbolHash;
467     }
}
```

Listing 3.15: UniswapAnchoredView::addNewConfig()

If the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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

Status The issue has been confirmed by the team.



4 | Conclusion

In this audit, we have analyzed the LEND by TEN Finance protocol design and implementation. The protocol is designed to be a money market that is inspired from Compound with the extensions of supporting shared revenue fees. During the audit, we notice that the current code base is well organized.

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] Fabian Vogelsteller And Vitalik Buterin. EIP-20: ERC-20 Token Standard. <https://eips.ethereum.org/EIPS/eip-20>.
- [2] Aislinn Keely. Cream Finance Exploited in \$18.8 million Flash Loan Attack. <https://www.theblockcrypto.com/linked/116055/creamfinance-exploited-in-18-8-million-flash-loan-attack>.
- [3] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [4] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [5] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.html>.
- [6] MITRE. CWE-682: Incorrect Calculation. <https://cwe.mitre.org/data/definitions/682.html>.
- [7] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [8] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [9] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.

- [10] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [11] MITRE. CWE CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.html>.
- [12] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. <https://cwe.mitre.org/data/definitions/389.html>.
- [13] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [14] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [15] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [16] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [17] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.