SMART CONTRACT AUDIT REPORT

for

UniLend V2

Prepared By: Yiqun Chen

PeckShield

January 10, 2022

Document Properties

Client	UniLend Finance
Title	Smart Contract Audit Report
Target	UniLend V2
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 10, 2022	Shulin Bie	Final Release
1.0-rc	December 12, 2021	Shulin Bie	Release Candidate

Contact

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

Name	Yiqun Chen
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 UniLend V2, 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 UniLend V2

UniLend V2 is a permission-less decentralized protocol that combines spot trading services and money markets with lending and borrowing services through smart contracts, which allows the users to unlock their token's functionality for lending to receive an interest rate and for borrowing by paying an interest rate. Additionally, UniLend V2 innovatively introduces flashloan feature in its lending platform. The basic information of UniLend V2 is as follows:

Table 1.1: Basic Information of UniLend V2

Item	Description
Target	UniLend V2
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	January 10, 2022

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

- https://github.com/UniLend/unilendv2.git (14f96a7)

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

- https://github.com/UniLend/unilendv2.git (155a2c8)

1.2 About PeckShield

PeckShield Inc. [9] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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) [7], 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 `UniLend V2` 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	1	■
High	3	■ ■ ■
Medium	1	■
Low	3	■ ■ ■
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, 3 high-severity vulnerabilities, 1 medium-severity vulnerability, and 3 low-severity vulnerabilities.

Table 2.1: Key UniLend V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improper Logic Of UnilendV2Pool::repay()	Business Logic	Fixed
PVE-002	Critical	Lack Of Health Factor Check In UnilendV2Pool::redeem()	Business Logic	Fixed
PVE-003	Low	Improper Event Information In UnilendV2Pool::lend()	Business Logic	Fixed
PVE-004	Low	Improper Logic Of UnilendV2Position::position()	Business Logic	Fixed
PVE-005	Low	Potential Reentrancy Risk In UnilendV2Core::lend()	Time and State	Fixed
PVE-006	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-007	High	Improper Accrue Interest Calculation During Lending	Business Logic	Fixed
PVE-008	High	Improper User Liquidation Price Array Management In UnilendV2Pool	Business Logic	Fixed

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 Improper Logic Of UnilendV2Pool::repay()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: UnilendV2Pool
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In the UniLend V2 protocol, we notice the ERC721 standard is introduced to identify the user in different lending/borrowing pools. Additionally, the positionData mapping in the UnilendV2Pool contract maintains the relationship between the NFT and the user assets in the pool. While examining the logics of the UnilendV2Pool contract, we notice there is an improper implementation of the repay() routine (used to repay the borrowed assets by the user in the pool) that needs to be improved.

To elaborate, we show below the related code snippet of the repay() function. At the beginning of the repay() function, it makes the query, i.e., pM storage _positionMt = positionData[1] (line 736), to retrieve the user assets in the pool according to the input _nftID parameter. However, we observe the constant 1 rather than _nftID is incorrectly used to retrieve the user assets in the pool, which directly undermines the repay() function design. Given this, we suggest to correct the implementation as below: pM storage _positionMt = positionData[_nftID] (line 736). Note other routines, i.e., redeem() and redeemUnderlying(), can also benefit from this improvement.

Moreover, in the repay() function, if the input amount parameter (the expected repay amount) is larger than 0, the borrowed token1 will be repaid, and vice versa. Additionally, if the input amount parameter is larger than the amount of the total borrowed assets, it will be re-assigned to the amount of the total borrowed assets. However, it comes to our attention that the amount is incorrectly re-assigned to -int(_totalLiability) rather than _totalLiability (line 771) during repaying token1, which doesn't meet the original design intention.

```
    function repay(uint _nftID, int amount, address _payer) external onlyCore returns(
        int _rAmount) {
        accrueInterest();

        pM storage _positionMt = positionData[1];

        if(amount < 0){
            tM storage _tm0 = token0Data;

            uint _totalBorrow = _tm0.totalBorrow;
            uint _totalLiability = getShareValue( _totalBorrow, _tm0.totalBorrowShare,
                _positionMt.token0borrowShare ) ;

            if(uint(-amount) > _totalLiability){
                amount = -int(_totalLiability);

                _burnBposition(_nftID, _positionMt.token0borrowShare, 0);

                _tm0.totalBorrow = _tm0.totalBorrow.sub(_totalLiability);
            }
            else {
                uint amountToShare = getShareByValue( _totalBorrow, _tm0.
                    totalBorrowShare, uint(-amount) );

                _burnBposition(_nftID, amountToShare, 0);

                _tm0.totalBorrow = _tm0.totalBorrow.sub(uint(-amount));
            }

            _rAmount = amount;

            emit RepayBorrow(token0, _nftID, uint(-amount), _tm0.totalBorrow, _payer);
        }

        if(amount > 0){
            tM storage _tm1 = token1Data;

            uint _totalBorrow = _tm1.totalBorrow;
            uint _totalLiability = getShareValue( _totalBorrow, _tm1.totalBorrowShare,
                _positionMt.token1borrowShare) ;

            if(uint(amount) > _totalLiability){
                amount = -int(_totalLiability);

                _burnBposition(_nftID, 0, _positionMt.token1borrowShare);

                _tm1.totalBorrow = _tm1.totalBorrow.sub(_totalLiability);
            }
            else {
                uint amountToShare = getShareByValue( _totalBorrow, _tm1.
                    totalBorrowShare, uint(amount) );
```

```
                _burnBposition(_nftID, 0, amountToShare);

                _tm1.totalBorrow = _tm1.totalBorrow.sub(uint(amount));
            }

            _rAmount = amount;

            emit RepayBorrow(token1, _nftID, uint(amount), _tm1.totalBorrow, _payer);
        }

        _updateUserLiquidationPrice(_nftID);
    }
```

Listing 3.1: `UnilendV2Pool::repay()`

Recommendation Correct the implementation of the routines as above-mentioned.

Status The issue has been addressed by the following commit: `ade19ba` and `935fd25`.

3.2 Lack Of Health Factor Check In UnilendV2Pool::redeem()

- ID: PVE-002
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `UnilendV2Pool`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In the `UniLend V2` protocol, the `UnilendV2Pool` contract plays a core role. In particular, one routine, i.e., `redeem()`, is designed to redeem the assets lent to the `lending/borrowing` pool before. While examining the logic of the `redeem()` routine, we notice there is no health factor check to keep the user account healthy in the pool.

To elaborate, we show below the related code snippet of the `redeem()` routine. By design, in the `lending/borrowing` pool, the value of the lent assets by the user should be larger than the value of the borrowed assets to keep the user account healthy. However, we notice there is a lack of health factor check in the `redeem()` function to keep the user account healthy in the pool. With that, the vulnerability can be exploited to drain the assets from the pool. Given this, we suggest to add health factor check to keep the user account healthy.

```
    function redeem(uint _nftID, int tok_amount, address _receiver) external onlyCore
        returns(int _amount) {
        accrueInterest();
```

```
        pM storage _positionMt = positionData[1];

        if(tok_amount < 0){
            require(_positionMt.token0lendShare >= uint(-tok_amount), "Balance Exceeds
                Requested");

            tM storage _tm0 = token0Data;

            uint tokenBalance0 = IERC20(token0).balanceOf(address(this));
            uint _totTokenBalance0 = tokenBalance0.add(_tm0.totalBorrow);
            uint poolAmount = getShareValue(_totTokenBalance0, _tm0.totalLendShare, uint
                (-tok_amount));

            _amount = -int(poolAmount);

            require(tokenBalance0 >= poolAmount, "Not enough Liquidity");

            _burnLPposition(_nftID, uint(-tok_amount), 0);

            transferToUser(token0, payable(_receiver), poolAmount);

            emit Redeem(token0, _nftID, uint(-tok_amount), poolAmount);
        }

        if(tok_amount > 0){
            require(_positionMt.token1lendShare >= uint(tok_amount), "Balance Exceeds
                Requested");

            tM storage _tm1 = token1Data;

            uint tokenBalance1 = IERC20(token1).balanceOf(address(this));
            uint _totTokenBalance1 = tokenBalance1.add(_tm1.totalBorrow);
            uint poolAmount = getShareValue(_totTokenBalance1, _tm1.totalLendShare, uint
                (tok_amount));

            _amount = int(poolAmount);

            require(tokenBalance1 >= poolAmount, "Not enough Liquidity");

            _burnLPposition(_nftID, 0, uint(tok_amount));

            transferToUser(token1, payable(_receiver), poolAmount);

            emit Redeem(token1, _nftID, uint(tok_amount), poolAmount);
        }

        _updateUserLiquidationPrice(_nftID);
    }
```

Listing 3.2: `UnilendV2Pool::redeem()`

Note other routines, i.e., `redeemUnderlying()` and `borrow()`, share the same issue.

Recommendation Add necessary health factor check in above-mentioned routines.

Status The issue has been addressed by the following commit: `a5f70ae`.

3.3 Improper Event Information In UnilendV2Pool::lend()

- ID: PVE-003
- Severity: Low
- Likelihood: High
- Impact: Low
- Target: `UnilendV2Pool/UnilendV2Core`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

While examining the events that reflect the `UnilendV2Pool` dynamics, we notice there is an incorrect event information in the `lend()` routine. To elaborate, we show below the related code snippet of the contract. By design, there are two different underlying tokens (i.e., `token0` and `token1`) in every `lending/borrowing` pool. The `lend()` routine is designed to lend the underlying token to the pool. If the input `amount` parameter of the `lend()` routine is larger than `0`, it means the user intends to lend `token1` to the pool, and vice versa. Meanwhile, the `event Lend( address indexed _asset, uint256 indexed _positionID, uint256 _amount, uint256 _token_amount )` is emitted to reflect the lending. However, we notice the third parameter of the event (line 583) is incorrect while the user lends `token1` to the pool. Given this, we suggest to correct the event as below: `emit Lend(token1, _nftID, uint(amount), ntokens1)` (line 583).

```
559     function lend(uint _nftID, int amount) external onlyCore returns(uint) {
560         accrueInterest();

562         uint ntokens0; uint ntokens1;

564         if(amount < 0){
565             tM storage _tm0 = token0Data;

567             uint tokenBalance0 = IERC20(token0).balanceOf(address(this));
568             uint _totTokenBalance0 = tokenBalance0.add(_tm0.totalBorrow);
569             ntokens0 = calculateShare(_tm0.totalLendShare, _totTokenBalance0.sub(uint(-
                    amount)), uint(-amount));
570             require(ntokens0 > 0, 'Insufficient Liquidity Minted');
```

```
            emit Lend(token0, _nftID, uint(-amount), ntokens0);
        }

        if(amount > 0){
            tM storage _tm1 = token1Data;

            uint tokenBalance1 = IERC20(token1).balanceOf(address(this));
            uint _totTokenBalance1 = tokenBalance1.add(_tm1.totalBorrow);
            ntokens1 = calculateShare(_tm1.totalLendShare, _totTokenBalance1.sub(uint(
                amount)), uint(amount));
            require(ntokens1 > 0, 'Insufficient Liquidity Minted');

            emit Lend(token1, _nftID, uint(amount), ntokens0);
        }

        _mintLPposition(_nftID, ntokens0, ntokens1);

        _updateUserLiquidationPrice(_nftID);

        return 0;
    }
```

Listing 3.3: `UnilendV2Pool::lend()`

Moreover, we notice there is a lack of emitting an event to reflect `governor` changes. In the following, we show below the related code snippet of the contract.

```
    function setGovernor(address _address) external onlyGovernor {
        require(_address != address(0), "UnilendV2: ZERO ADDRESS");
        governor = _address;
    }
```

Listing 3.4: `UnilendV2Core::setGovernor()`

With that, we suggest to add a new event `NewGovernor` whenever the new `governor` is changed. Also, the new `governor` information is better `indexed`. Note each emitted event is represented as a topic that usually consists of the signature (from a `keccak256` hash) of the event name and the types (`uint256`, `string`, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the `governor` information is typically queried, it is better treated as a topic, hence the need of being `indexed`.

Recommendation Properly emit the above-mentioned events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been addressed by the following commits: `28af721` and `3f1fae8`.

3.4 Improper Logic Of UnilendV2Position::position()

- ID: PVE-004
- Severity: Low
- Likelihood: High
- Impact: Low
- Target: UnilendV2Position
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

By design, the UniLend V2 protocol introduces the ERC721 standard to identify the user in different lending/borrowing pools. The UnilendV2Position contract implements the ERC721 standard and maintains the relationship between the NFT and the user information. While examining the logics of the contract, we observe an improper implementation of the position() routine that can be improved safely.

To elaborate, we show below the related code snippet of the UnilendV2Position contract. The position() routine is designed to retrieve the assets of the user (specified by the input _nftID) in the pool, including token0 lending/borrowing assets and token1 lending/borrowing assets. However, we notice the pool.userBalanceOftoken0(_nftID) (line 560) is incorrectly called to query the token1 assets. Given this, we suggest to correct the implementation as below: (_positionData.lendBalance1 , _positionData.borrowBalance1)= pool.userBalanceOftoken1(_nftID) (line 560).

```
552     function position(uint _nftID) external view returns (nftPositionData memory
            _positionData){
553         if(nftPool[_nftID] != address(0)){
554             IUnilendV2PoolData pool = IUnilendV2PoolData(nftPool[_nftID]);

556             _positionData.token0 = pool.token0();
557             _positionData.token1 = pool.token1();

559             (_positionData.lendBalance0, _positionData.borrowBalance0) = pool.
                    userBalanceOftoken0(_nftID);
560             (_positionData.lendBalance1, _positionData.borrowBalance1) = pool.
                    userBalanceOftoken0(_nftID);
561         }
562     }
```

Listing 3.5: UnilendV2Position::position()

Recommendation Correct the implementation of the position() routine as above-mentioned.

Status The issue has been addressed by the following commit: a2325df.

3.5 Potential Reentrancy Risk In UnilendV2Core::lend()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact:Medium
- Target: `UnilendV2Core`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [2]

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` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

In the `UnilendV2Core` contract, we notice there is a routine (i.e., `lend()`) that has potential reentrancy risk. To elaborate, we show below the related code snippet of the `lend()` routine in the `UnilendV2Core` contract. In the `lend()` function, the internal `iLend()` function is called to deposit the assets to the pool. While examining the logic of the internal `iLend()` function, we notice the `IERC20(_token).safeTransferFrom(_user, _pool, uint(-_amount))` is called (line 471) to transfer the `_token` to the `UnilendV2Core` contract. If the `_token` faithfully implements the ERC777-like standard, then the `lend()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

```
453     function lend(address _pool, int _amount) external onlyAmountNotZero(_amount)
            returns(uint mintedTokens) {
454         (address _token0, address _token1) = getPoolTokens(_pool);
455         require(_token0 != address(0), 'UnilendV2: POOL NOT FOUND');
456
457         uint _nftID = IUnilendV2Position(positionsAddress).getNftId(_pool, msg.sender);
458         if(_nftID == 0){
459             _nftID = IUnilendV2Position(positionsAddress).newPosition(_pool, msg.sender)
                    ;
460         }
461
462         address _reserve = _amount < 0 ? _token0 : _token1;
463         mintedTokens = iLend(_pool, _reserve, _amount, _nftID);
464     }
465
466     function iLend(address _pool, address _token, int _amount, uint _nftID) internal
            returns(uint mintedTokens) {
467         address _user = msg.sender;
468
```

```
469         if(_amount < 0){
470             uint reserveBalance = IERC20(_token).balanceOf(_pool);
471             IERC20(_token).safeTransferFrom(_user, _pool, uint(-_amount));
472             _amount = -int( ( IERC20(_token).balanceOf(_pool) ).sub(reserveBalance) );
473         }
474 
475         if(_amount > 0){
476             uint reserveBalance = IERC20(_token).balanceOf(_pool);
477             IERC20(_token).safeTransferFrom(_user, _pool, uint(_amount));
478             _amount = int( ( IERC20(_token).balanceOf(_pool) ).sub(reserveBalance) );
479         }
480 
481         mintedTokens = IUnilendV2Pool(_pool).lend(_nftID, _amount);
482     }
```

Listing 3.6: `UnilendV2Core::lend()&&iLend()`

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend()` and `tokensReceived()` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In our case, the above hook can be planted in `IERC20(_token).safeTransferFrom(_user, _pool, uint(-_amount))` (line 471). By doing so, we can effectively keep `reserveBalance` intact (used for the calculation of actual `_token` amount transferred to the `UnilendV2Core` at line 472). With a lower `reserveBalance`, the re-entered `UnilendV2Core::lend()` is able to obtain more lending credits. It can be repeated to exploit this vulnerability for gains, just like earlier `Uniswap/Lendf.Me` hack [10].

Note that other functions, i.e., `redeem()`, `redeemUnderlying()`, `borrow()`, `repay()` and `liquidate()`, can also benefit from the reentrancy protection.

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks.

Status The issue has been addressed by the following commit: `7fdcb0c`.

3.6 Trust Issue Of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: UnilendV2Core
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the UniLend V2 protocol, there is a privileged governor account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure oracle address). In the following, we show the representative functions potentially affected by the privileged governor.

```
    function setPositionAddress(address _address) external onlyGovernor {
        require(_address != address(0), "UnilendV2: ZERO ADDRESS");
        positionsAddress = _address;
    }

    /**
    * @dev set new oracle address.
    * @param _address new address
    **/
    function setOracleAddress(address _address) external onlyGovernor {
        require(_address != address(0), "UnilendV2: ZERO ADDRESS");
        oracleAddress = _address;
    }
```

Listing 3.7: UnilendV2Core::setPositionAddress()&&setOracleAddress()

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged governor account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the UniLend V2 design.

Recommendation Promptly transfer the privileged governor 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 This issue has been confirmed by the team.

3.7 Improper Accrue Interest Calculation During Lending

- ID: PVE-007
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: UnilendV2Core/UnilendV2Pool
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

By design, the UnilendV2Core contract is the main entry for interaction with users. In particular, one routine, i.e., UnilendV2Core::lend(), is designed to lend the supported token to the lending/borrowing pool. While examining the process of lending token, we notice the accrue interest calculation should be improved.

To elaborate, we show below the related code snippet of the UnilendV2Core/UnilendV2Pool contracts. In the internal UnilendV2Core::iLend() function, we notice the _token is transferred into the lending/borrowing pool firstly (line 471), and then the UnilendV2Pool::lend() is called (line 481) to update the pool status. In the UnilendV2Pool::lend() function, the accrueInterest() is called (line 560) to calculate the accrue interest. However, we observe IERC20(token0).balanceOf(address(this)) (line 212) is used during calculating the accrue interest, which results in inaccurate interest calculation because of the incorrect inclusion of the newly transferred token. Given this, we suggest to transfer the _token to the lending/borrowing pool at the end of the UnilendV2Core::iLend() function.

```
453     function lend(address _pool, int _amount) external onlyAmountNotZero(_amount)
            returns(uint mintedTokens) {
454         (address _token0, address _token1) = getPoolTokens(_pool);
455         require(_token0 != address(0), 'UnilendV2: POOL NOT FOUND');

457         uint _nftID = IUnilendV2Position(positionsAddress).getNftId(_pool, msg.sender);
458         if(_nftID == 0){
459             _nftID = IUnilendV2Position(positionsAddress).newPosition(_pool, msg.sender)
                    ;
460         }

462         address _reserve = _amount < 0 ? _token0 : _token1;
463         mintedTokens = iLend(_pool, _reserve, _amount, _nftID);
464     }

466     function iLend(address _pool, address _token, int _amount, uint _nftID) internal
            returns(uint mintedTokens) {
467         address _user = msg.sender;

469         if(_amount < 0){
470             uint reserveBalance = IERC20(_token).balanceOf(_pool);
471             IERC20(_token).safeTransferFrom(_user, _pool, uint(-_amount));
```

```
            _amount = -int( ( IERC20(_token).balanceOf(_pool) ).sub(reserveBalance) );
        }

        if(_amount > 0){
            uint reserveBalance = IERC20(_token).balanceOf(_pool);
            IERC20(_token).safeTransferFrom(_user, _pool, uint(_amount));
            _amount = int( ( IERC20(_token).balanceOf(_pool) ).sub(reserveBalance) );
        }

        mintedTokens = IUnilendV2Pool(_pool).lend(_nftID, _amount);
    }
```

Listing 3.8: `UnilendV2Pool::lend()`

```
    function lend(uint _nftID, int amount) external onlyCore returns(uint) {
        accrueInterest();

        ...
    }
```

Listing 3.9: `UnilendV2Pool::lend()`

```
    function accrueInterest() public {
        uint remainingBlocks = block.number - lastUpdated;

        if(remainingBlocks > 0){
            tM storage _tm0 = token0Data;
            tM storage _tm1 = token1Data;

            uint interestRate0 = getInterestRate0(_tm0.totalBorrow,
                getAvailableLiquidity0());
            uint interestRate1 = getInterestRate1(_tm1.totalBorrow,
                getAvailableLiquidity1());

            _tm0.totalBorrow = _tm0.totalBorrow.add( calculateInterest(_tm0.totalBorrow,
                 interestRate0, remainingBlocks) );
            _tm1.totalBorrow = _tm1.totalBorrow.add( calculateInterest(_tm1.totalBorrow,
                 interestRate1, remainingBlocks) );

            lastUpdated = block.number;

            emit InterestUpdate(interestRate0, interestRate1, _tm0.totalBorrow, _tm1.
                totalBorrow);
        }
    }
```

Listing 3.10: `UnilendV2Pool::accrueInterest()`

```
    function getAvailableLiquidity0() public view returns (uint _available) {
        tM memory _tm0 = token0Data;

        uint totalBorrow = _tm0.totalBorrow;
```

```
        uint totalLiq = totalBorrow.add( IERC20(token0).balanceOf(address(this)) );
        uint maxAvail = ( totalLiq.mul( uint(100).sub(rf) ) ).div(100);

        if(maxAvail > totalBorrow){
            _available = maxAvail.sub(totalBorrow);
        }
    }

    function getAvailableLiquidity1() public view returns (uint _available) {
        tM memory _tm1 = token1Data;

        uint totalBorrow = _tm1.totalBorrow;
        uint totalLiq = totalBorrow.add( IERC20(token1).balanceOf(address(this)) );
        uint maxAvail = ( totalLiq.mul( uint(100).sub(rf) ) ).div(100);

        if(maxAvail > totalBorrow){
            _available = maxAvail.sub(totalBorrow);
        }
    }
```

Listing 3.11: `UnilendV2Pool::getAvailableLiquidity0()`

Recommendation Correct the implementation of the `UnilendV2Core::iLend()` routine as above-mentioned.

Status The issue has been addressed by the following commits: `935fd25` and `caa34fd`.

3.8 Improper User Liquidation Price Array Management In UnilendV2Pool

- ID: PVE-008
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `UnilendV2Pool`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

By design, the `UniLend V2` protocol provides a well-designed `liquidation` mechanism, which groups the users according to the collateral ratio of the lent assets divided by the borrowed assets with `1/1000` accuracy. In particular, one routine, i.e., `_updateUserLiquidationPrice()`, is designed to timely update the user liquidation group once the user assets changes in the `lending/borrowing` pool. While examining the logic of it, we observe there is an improper implementation that needs to be improved. To elaborate, we show below the related code snippet of the `_updateUserLiquidationPrice()` routine.

To illustrate, if Bob's collateral ratio changes, his information will be removed from the old liquidation group and added to the new liquidation group (lines 493 - 518). Because the user information is managed by using array (e.g., `mapping(uint => uint[])public liquidationPrices0`), we observe the element of the array that holds Bob's information is replaced by the last element of the array and the last element is reset with `delete` (lines 500 - 506). However, it ignores a special situation where Bob's information is stored in the last element of the array (line 500), which will result that Bob's information cannot be removed from the old liquidation group.

Moreover, we emphasize that `delete` just resets the storage rather than releases the storage. Given this, we suggest to use `pop()` to release the storage. Meanwhile, the `liquidateUser0()` and `liquidateUser1()` routines can also benefit from the improvement.

```
483     function _updateUserLiquidationPrice(uint _nftID) internal {
484         (uint _lendBalance0, uint _borrowBalance0) = userBalanceOftoken0(_nftID);
485         (uint _lendBalance1, uint _borrowBalance1) = userBalanceOftoken1(_nftID);


488         if(_borrowBalance0 > 0){
489             uint _estLendAfterLb = ( _lendBalance1.mul(uint(100).sub(lb)) ).div(100);
490             uint _userLiquidationPrice = priceScaled( _estLendAfterLb.mul(10**18).div(
                    _borrowBalance0) );
491             uint _userLqIndex = userLiquidationIndex0[_nftID];

493             if(_userLiquidationPrice0[_nftID] != _userLiquidationPrice){

495                 // remove user index and update last one
496                 uint _lastUserLqPrice;
497                 if(_userLqIndex > 0){
498                     _lastUserLqPrice = _userLiquidationPrice0[_nftID];
499                     uint _lastIndexforLastPrice = liquidationPrices0[_lastUserLqPrice].
                            length - 1;
500                     if(_userLqIndex < _lastIndexforLastPrice){
501                         uint _lastLqNft = liquidationPrices0[_lastUserLqPrice][
                                _lastIndexforLastPrice];

503                         userLiquidationIndex0[_lastLqNft] = _userLqIndex;
504                         liquidationPrices0[_lastUserLqPrice][_userLqIndex] = _lastLqNft;
505                         delete liquidationPrices0[_lastUserLqPrice][
                                _lastIndexforLastPrice];
506                     }
507                 }

509                 if(liquidationPrices0[_userLiquidationPrice].length == 0){
510                     liquidationPrices0[_userLiquidationPrice].push(0);
511                 }

513                 liquidationPrices0[_userLiquidationPrice].push(_nftID);
514                 userLiquidationIndex0[_nftID] = liquidationPrices0[_userLiquidationPrice
                        ].length - 1;
515                 _userLiquidationPrice0[_nftID] = _userLiquidationPrice;
```

```
                emit LiquidationPriceUpdate(_nftID, _userLiquidationPrice,
                    _lastUserLqPrice, _estLendAfterLb);
            }
        }


        if(_borrowBalance1 > 0){
            uint _estLendAfterLb = ( _lendBalance0.mul(uint(100).sub(lb)).div(100) );
            uint _userLiquidationPrice = priceScaled( _borrowBalance1.mul(10**18).div(
                _estLendAfterLb) );
            uint _userLqIndex = userLiquidationIndex1[_nftID];

            if(_userLiquidationPrice1[_nftID] != _userLiquidationPrice){

                // remove user index and update last one
                uint _lastUserLqPrice;
                if(_userLqIndex > 0){
                    _lastUserLqPrice = _userLiquidationPrice1[_nftID];
                    uint _lastIndexforLastPrice = liquidationPrices1[_lastUserLqPrice].
                        length - 1;
                    if(_userLqIndex < _lastIndexforLastPrice){
                        uint _lastLqNft = liquidationPrices1[_lastUserLqPrice][
                            _lastIndexforLastPrice];

                        userLiquidationIndex1[_lastLqNft] = _userLqIndex;
                        liquidationPrices1[_lastUserLqPrice][_userLqIndex] = _lastLqNft;
                        delete liquidationPrices1[_lastUserLqPrice][
                            _lastIndexforLastPrice];
                    }
                }

                if(liquidationPrices1[_userLiquidationPrice].length == 0){
                    liquidationPrices1[_userLiquidationPrice].push(0);
                }

                liquidationPrices1[_userLiquidationPrice].push(_nftID);
                userLiquidationIndex1[_nftID] = liquidationPrices1[_userLiquidationPrice
                    ].length - 1;
                _userLiquidationPrice1[_nftID] = _userLiquidationPrice;

                emit LiquidationPriceUpdate(_nftID, _userLiquidationPrice,
                    _lastUserLqPrice, _estLendAfterLb);
            }
        }
    }
```

Listing 3.12: `UnilendV2Pool::_updateUserLiquidationPrice()`

Recommendation Correct the implementation of the above-mentioned routines.

Status The issue has been addressed by the following commits: `2c4e3f8` and `155a2c8`.

4 | Conclusion

In this audit, we have analyzed the `UniLend V2` design and implementation. `UniLend V2`, as a permissionless decentralized protocol, supports lending and borrowing services through smart contracts. The users have the capability to unlock their token's functionality for lending to receive an interest rate and for borrowing by paying an interest rate. Additionally, `UniLend V2` innovatively introduces `flashloan` feature in lending platform. It enriches the `UniLend Finance` ecosystem. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

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

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

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

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

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

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

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

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

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

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.