SMART CONTRACT AUDIT REPORT

for

Themis V3

Prepared By: Yiqun Chen

Hangzhou, China

January 11, 2022

Document Properties

Client	Themis Protocol
Title	Smart Contract Audit Report
Target	Themis V3
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 11, 2022	Shulin Bie	Final Release
1.0-rc2	January 7, 2022	Shulin Bie	Release Candidate #2
1.0-rc1	December 14, 2021	Shulin Bie	Release Candidate #1

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 `Themis V3` 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 Themis V3

`Themis V3` is a permission-less decentralized protocol that provides lending and borrowing services through smart contracts, which allows the users to lend assets to receive an interest rate and borrow assets with `UniswapV3 positions` as collateral. Compared to other P2P lending protocols, `Themis V3` allows users to create anonymous lending between pools of funds and mortgagers of `NFTs`, including `UniswapV3 positions`. Market makers can obtain market-making benefits and form loan settlement relationships with the capital pool to borrow crypto assets for other purposes.

Table 1.1: Basic Information of Themis V3

Item	Description
Target	Themis V3
Website	*https://themis.exchange/*
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 11, 2022

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

- https://github.com/Themis-protocol/audit-contract.git (7643a87)

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

- https://github.com/Themis-protocol/audit-contract.git (c8f2830)

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

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.

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) [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 `Themis V3` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	3	■■■
Low	3	■■■
Informational	0	
Undetermined	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 2 high-severity vulnerabilities, 3 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key Themis V3 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improper Design For ThemisEarlyFarmingNFT(ERC721) Transfer	Business Logic	Fixed
PVE-002	Medium	Timely Update Pool Reward During Pool Weight Changes	Business Logic	Fixed
PVE-003	Low	Potential DoS In ThemisAuction::doAuction()	Time and State	Confirmed
PVE-004	High	Improper Borrow Interest Calculation In userReturn()	Business Logic	Fixed
PVE-005	Medium	Improper Logic Of settlementBorrow()	Business Logic	Fixed
PVE-006	Low	Duplicate Pool Detection and Prevention	Business Logic	Fixed
PVE-007	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-008	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed

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 Design For ThemisEarlyFarmingNFT(ERC721) Transfer

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

Description

In the `Themis V3` protocol, the `ThemisEarlyFarming` contract implements a term deposit mechanism. With that, the user can not only earn lending interests on his deposits, but also get bonus `rewardToken` . Additionally, the `ThemisEarlyFarmingNFT` contract implements the `ERC721` standard to identify each deposit through the `ThemisEarlyFarming` contract. In other words, each deposit is treated as a `NFT` and its related information is stored in the `earlyFarmingNftInfos` and `userNftPeriodPoolIdAllTokenIds` mapping (lines 27/29) of the `ThemisEarlyFarmingNFT` contract. In particular, one `ERC721` standard interface, i.e., `safeTransferFrom()`, is overridden to support `NFT` transfer with its related deposit information update. While examining the logic of it, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `ThemisEarlyFarming` and `ThemisEarlyFar-mingNFT` contracts. In the overridden `safeTransferFrom()` function of the `ThemisEarlyFarmingNFT` contract, the internal `_updateNftInfo()` function is called (line 169) to update the `NFT` related deposit information. However, it ignores the fact that the `userPeriodInfos` mapping of the `ThemisEarlyFarming` contract also stores the deposit information. The calling of `safeTransferFrom()` will introduce unexpected state inconsistencies between the `ThemisEarlyFarming` and `ThemisEarlyFarmingNFT` contracts, which may result withdrawal failure for the `NFT` corresponding deposit.

Moreover, note that the lending interests and bonus `rewardToken` will be transferred to the new owner with the `NFT` transfer.

```
contract ThemisEarlyFarmingNFT is IThemisEarlyFarmingNFTStorage,ERC721,
    ERC721Enumerable,Governance {
    using Counters for Counters.Counter;
    using SafeMath for uint256;
    using EnumerableSet for EnumerableSet.UintSet;

    event WithdarwAmountsEvent(address indexed sender,WtihdrawNftParams
        wtihdrawNftParams);
    Counters.Counter private _tokenIdCounter;

    address public miner;
    address public themisEarlyFarmingNFTDescriptor;

    mapping(uint256 => EarlyFarmingNftInfo) public earlyFarmingNftInfos;

    mapping(address => mapping(uint256 =>EnumerableSet.UintSet))
        userNftPeriodPoolIdAllTokenIds; // user address => periodPoolId =>
        periodPoolId set // all records

    ...
}
```

Listing 3.1: `ThemisEarlyFarmingNFT`

```
function safeTransferFrom(
    address from,
    address to,
    uint256 tokenId,
    bytes memory _data
) public virtual override(ERC721) {
    require(_isApprovedOrOwner(_msgSender(), tokenId), "ERC721: transfer caller is
        not owner nor approved");
    _updateNftInfo(tokenId,to);
    super._safeTransfer(from, to, tokenId, _data);
}

function _updateNftInfo(uint256 tokenId,address to) internal{
    EarlyFarmingNftInfo storage nftInfo = earlyFarmingNftInfos[tokenId];
    address nftOnwer = nftInfo.onwerUser;
    userNftPeriodPoolIdAllTokenIds[nftOnwer][nftInfo.periodPoolId].remove(tokenId);
    nftInfo.onwerUser = to;
    userNftPeriodPoolIdAllTokenIds[to][nftInfo.periodPoolId].add(tokenId);
}
```

Listing 3.2: `ThemisEarlyFarmingNFT::safeTransferFrom()`

```
function userDeposit(uint256 _periodPoolId,uint256 _amount) external
    checkPeriodVaild(_periodPoolId) {
    require(_amount > 0, "deposit input invalid amount.");
```

```
        address _user = msg.sender;

        _settlementProfit(_periodPoolId);

        uint256 _currBlock = block.number;

        PeriodPool storage _periodPool = periodPools[_periodPoolId];
        address _token = _periodPool.token;
        // update gobal
        tokenCurrTotalDeposit[_token] = tokenCurrTotalDeposit[_token].add(_amount);
        // update period pool
        _periodPool.currTotalDeposit = _periodPool.currTotalDeposit.add(_amount);
        // update user

        UserPeriodInfo storage _userPeriodInfo = userPeriodInfos[_periodPoolId][_user];
        _userPeriodInfo.currDeposit = _userPeriodInfo.currDeposit.add(_amount);

        IERC20(_token).safeTransferFrom(_user, address(this), _amount);

        IERC20(_token).safeApprove(address(themisLendCompound),_amount);

        themisLendCompound.userLend(_periodPool.lendPoolId,_amount);

        //send user a NFT
        EarlyFarmingNftInfo memory _nftInfo = EarlyFarmingNftInfo({
            periodPoolId:_periodPoolId,
            buyUser:_user,
            onwerUser: _user,
            pledgeToken: _token,
            pledgeAmount: _amount,
            withdrawAmount: 0,
            lastUnlockBlock: _currBlock,
            startBlock: _currBlock,
            endBlock: _currBlock + _periodPool.periodBlock,
            buyTime : block.timestamp,
            perBlockUnlockAmount:0,
            lastInterestsShare: _periodPool.interestsShare,
            lastRewardsShare: _periodPool.rewardsShare
        });

        themisEarlyFarmingNFT.safeMint(_user,_nftInfo);


        emit UserDepositEvent(_user,_periodPoolId,_amount);
    }
```

Listing 3.3: `ThemisEarlyFarming::userDeposit()`

Recommendation Correct the implementation of the `safeTransferFrom()` routine to keep state consistency between the `ThemisEarlyFarming` and `ThemisEarlyFarmingNFT` contracts.

Status The issue has been addressed by the following commits: `ee75e72` and `9ee68dd`.

3.2 Timely Update Pool Reward During Pool Weight Changes

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: ThemisEarlyFarming
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The ThemisEarlyFarming contract provides an incentive mechanism that rewards the staking of supported assets with the rewardToken token. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of tokens in the reward pool.

The staking pools can be dynamically added via addPeriodPool() and the weights of supported pools can be adjusted via setPeriodAllocPoint(). When analyzing the pool weight update routine setPeriodAllocPoint(), we notice the need of timely invoking _updatePeriodRewardShare() for all the staking pools to update the reward distribution before the new pool weight becomes effective.

```
    function setPeriodAllocPoint(uint256 _periodPoolId,uint256 _allocPoint,bool _misUpdate) external onlyGovernance{
        //todo update all period
        PeriodPool storage _periodPool = periodPools[_periodPoolId];
        uint256 _beforeAllocPoint = _periodPool.allocPoint;
        _periodPool.allocPoint = _allocPoint;
        totalAllocPoint = totalAllocPoint.add(_allocPoint).sub(_beforeAllocPoint);
        if(!_misUpdate){
            for(uint256 i=0;i<periodPools.length;i++){
                _updatePeriodRewardShare(i);
            }
        }
        emit SetPeriodAllocPointEvent(msg.sender,_periodPoolId,_beforeAllocPoint,_allocPoint);
    }
```

Listing 3.4: ThemisEarlyFarming::setPeriodAllocPoint()

If the call to _updatePeriodRewardShare() is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens.

Recommendation Timely invoke _updatePeriodRewardShare() for all the staking pools when any pool's weight has been updated.

Status The issue has been addressed by the following commit: 9ee68dd.

3.3 Potential DoS In ThemisAuction::doAuction()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `ThemisAuction`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [2]

Description

By design, the `Themis V3` protocol provides three indispensable functions (i.e., `lend`, `borrow`, and `liquidate`), which allow the user deposits the `UniswapV3 position` as collateral to borrow assets. When a borrower's collateral `UniswapV3 position` reaches the liquidated criterion, it will be transferred to the `ThemisAuction` contract (that implements the generic English auction) to do auction. After that, the winner of the auction will liquidate the `UniswapV3 position`. In particular, one entry routine, i.e., `doAuction()`, is designed to bid for the `UniswapV3 position`. While examining its logic, we observe it is exposed to potential `DoS` risks.

To elaborate, we show below the code snippet of the `doAuction()` routine in the `ThemisAuction` contract. In the `doAuction()` routine, we notice `_payToken.safeTransfer(_returnRecord.auctionUser, _returnRecord.auctionAmount)` (line 167) is called to refund the `_payToken` to the last bidder if the new bid price is larger than the last one. If the `_payToken` faithfully implements the ERC777-like standard, then the `doAuction()` routine is exposed to `DoS` vulnerability and this risk needs to be properly mitigated.

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 `_payToken.safeTransfer(_returnRecord.auctionUser,_returnRecord.auctionAmount)` (line 167). In the hook, the hacker can always revert the transaction. By doing so, he/she can win the auction eventually, which directly undermines the assumption of the `Themis V3` design.

```
130     function doAuction(uint256 auctionId,uint256 amount) external{
131         require(_holderAuctioningIds.contains(auctionId),"This auction not exist.");
132         (uint256 _auctionAmount,uint256 _onePrice,uint256 _remainTime,,,,) =
                _getCurrSaleInfo(auctionId,false);
```

```
        require(_remainTime >0,"Over time.");

        AuctionInfo storage _auctionInfo = auctionInfos[auctionId];
        require(_auctionInfo.state == 0  _auctionInfo.state == 1,"This auction state error.");

        require(amount >= _auctionInfo.laestBidPrice,"Must be greater than the existing maximum bid.");

        _auctionAmount = amount;



        IERC20 _payToken = IERC20(_auctionInfo.auctionToken);
        uint256 _addrBalance = _payToken.balanceOf(msg.sender);
        require(_addrBalance >= _auctionAmount,"Insufficient token amount.");

        AuctionRecord[] storage _auctionRecords = auctionRecords[auctionId];
        _auctionRecords.push(AuctionRecord({
            auctionUser: msg.sender,
            auctionAmount: _auctionAmount,
            blockTime: block.timestamp,
            returnPay: false
        }));

        _auctionInfo.laestBidPrice = amount;
        _auctionInfo.state = 1;

        _holderBidAuctionIds.add(auctionId);

        _payToken.safeTransferFrom(msg.sender,address(this),_auctionAmount);

        if(_auctionRecords.length > 1){
            AuctionRecord storage _returnRecord = _auctionRecords[_auctionRecords.length-2];
            if(!_returnRecord.returnPay ){
                _returnRecord.returnPay = true;
                _payToken.safeTransfer(_returnRecord.auctionUser,_returnRecord.auctionAmount);
            }
        }

        if(_auctionAmount >= _onePrice){
            _doHarvestAuction(auctionId,true);
        }


        emit DoAuctionEvent(_auctionInfo.bid,_auctionInfo.tokenId,auctionId,_auctionAmount,msg.sender);
    }
```

Listing 3.5: `ThemisAuction::doAuction()`

Recommendation One possible mitigation is to regulate the set of tokens that are permitted into `Themis V3`. In `Themis V3`, it is indeed possible to effectively regulate the set of tokens that can be supported.

Status The issue has been confirmed by the team. The team decides to leave it as ERC777 tokens will not be used in the protocol.

3.4 Improper Borrow Interest Calculation In userReturn()

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

Description

By design, the `ThemisBorrowCompound` contract is one of the main entries for interaction with users. In particular, one entry routine, i.e., `userReturn()`, is used by the borrower to repay the borrowed assets, in order to redeem the collateral `UniswapV3 position`. While examining its logic, we notice the borrowed interests calculation needs to be improved.

To elaborate, we show below the related code snippet of the `ThemisBorrowCompound` contract. In the `userReturn()` function, the borrowed interests is calculated as below: `uint256 _borrowInterests = _borrowPool.globalBowShare.sub(_borrowInfo.startBowShare).mul(_borrowInfo.amount).div(1e12)` (line 244). We notice the `_borrowPool.globalBowShare` is not updated to the latest, but is updated in the `_upGobalBorrowInfo()` function (line 256) called inside the `userReturn()` function subsequently, which results inaccurate borrowed interests calculation. Given this, we suggest to update the `_borrowPool.globalBowShare` to the latest before calculating the borrowed interests.

```
234     function userReturn(uint256 bid) public {
235         require(!isBorrowOverdue(bid), 'borrow is overdue');
236         BorrowInfo storage _borrowInfo = borrowInfo[bid];
237         require(_borrowInfo.user == msg.sender, 'not owner');

239         CompoundBorrowPool memory _borrowPool = borrowPoolInfo[_borrowInfo.pid];

241         BorrowUserInfo storage _user = borrowUserInfos[msg.sender][_borrowInfo.pid];


244         uint256 _borrowInterests = _borrowPool.globalBowShare.sub(_borrowInfo.
                startBowShare).mul(_borrowInfo.amount).div(1e12);

246         uint256 _totalReturn = _borrowInfo.amount.add(_borrowInterests);
```

```
        uint256 _userBalance = IERC20(_borrowPool.token).balanceOf(msg.sender);
         require(_userBalance >= _totalReturn, 'not enough amount.');

        _borrowInfo.returnBlock = block.number;
        _borrowInfo.interests = _borrowInterests;
        _borrowInfo.state = 2;


        _upGobalBorrowInfo(_borrowInfo.pid,_borrowInfo.amount,2);


        _updateRealReturnInterest(_borrowInfo.pid,_borrowInterests);


        uniswapV3.transferFrom(address(this), msg.sender, _borrowInfo.tokenId);


        _user.currTotalBorrow = _user.currTotalBorrow.sub(_borrowInfo.amount);
        _holderBorrowIds[msg.sender][_borrowInfo.pid].remove(bid);

        if(_user.currTotalBorrow == 0){
            _holderBorrowPoolIds[msg.sender].remove(_borrowInfo.pid);
        }

        IERC20(_borrowPool.token).safeTransferFrom(msg.sender, address(this),
            _totalReturn);
        IERC20(_borrowPool.token).safeTransfer(address(lendCompound),_borrowInfo.amount)
            ;

        emit UserReturn(msg.sender, bid,_borrowInfo.pid, _borrowInfo.amount,
            _borrowInterests);
    }
```

Listing 3.6: `ThemisBorrowCompound::userReturn()`

Note other routines, i.e., `transferToAuction()`, `isBorrowOverdue()`, and `pendingReturnInterests()`, share the same issue.

Recommendation Correct the implementation of the above-mentioned routines.

Status The issue has been addressed by the following commit: `368c0d9`.

3.5 Improper Logic Of settlementBorrow()

- ID: PVE-005
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `ThemisBorrowCompound`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As section 3.3 mentioned, when a borrower's collateral `UniswapV3 position` reaches the liquidated criterion, it will be transferred to the `ThemisAuction` contract with the calling of `transferToAuction()` to do auction. The winner will liquidate the borrower's collateral `UniswapV3 position`. And then the `settlementBorrow()` will be called to repay the borrowed assets of the borrower to the `lending/borrowing` pool. While examining the logic of it, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `ThemisBorrowCompound` contract. In the `settlementBorrow()` function, we notice the borrower related information stored in the `_holderBorrowIds` and `_holderBorrowPoolIds` mapping are removed (lines 392 - 396). However, we notice the same information has been removed before during the call to `transferToAuction()`. With that, we suggest to remove the logic from the `settlementBorrow()` function safely.

```
373     function settlementBorrow(uint256 bid) public {
374         BorrowInfo storage _borrowInfo = borrowInfo[bid];
375         require(_borrowInfo.state == 9, 'error status');


378         CompoundBorrowPool storage _borrowPool = borrowPoolInfo[_borrowInfo.pid];
379         BorrowUserInfo storage _user = borrowUserInfos[_borrowInfo.user][_borrowInfo.pid
            ];

381         _borrowInfo.state = 8;
382         _borrowInfo.returnBlock = block.number;

384         uint256 totalRuturn = _borrowInfo.amount.add(_borrowInfo.interests);



388         _upGobalBorrowInfo(_borrowInfo.pid,_borrowInfo.amount,2);

390         _updateRealReturnInterest(_borrowInfo.pid,_borrowInfo.interests);

392         _holderBorrowIds[_borrowInfo.user][_borrowInfo.pid].remove(bid);

394         if(_user.currTotalBorrow == 0){
395             _holderBorrowPoolIds[_borrowInfo.user].remove(_borrowInfo.pid);
```

```
        }

        IERC20(_borrowPool.token).safeTransferFrom(msg.sender, address(this),
            totalRuturn);

        IERC20(_borrowPool.token).safeTransfer(address(lendCompound),_borrowInfo.amount)
            ;

        emit SettlementBorrowEvent(bid, _borrowInfo.pid,_borrowInfo.amount,_borrowInfo.
            interests);
    }
```

Listing 3.7: `ThemisBorrowCompound::settlementBorrow()`

Recommendation Remove the redundant code from the `settlementBorrow()` routine safely.

Status The issue has been addressed by the following commits: `368c0d9` , `3042d51` and `e8006e6`.

3.6 Duplicate Pool Detection and Prevention

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ThemisLendCompound`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

By design, the `ThemisLendCompound` contract implements a `lending` mechanism, which allows the user to deposit the supported assets to earn lending interests. In current implementation, there are a number of concurrent pools that support different tokens and more can be scheduled for addition (via a proper governance procedure or moderated by a privileged account). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new `lending/borrowing` pools.

The addition of a new pool is implemented in `addPool()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `onlyGovernance`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    function addPool(ERC20 _token, uint256 _allocpoint, bool _withUpdate) public
        onlyGovernance {
        require(address(_token) != address(0), "_token is the zero address");

```

```
        address _spToken = _createToken("Supply-Provider Token", tokenPrefix.concat(_token.symbol()), _token.decimals());
        lendPoolInfo.push(
            CompoundLendPool({
                token: address(_token),
                spToken: _spToken,
                curSupply: 0,
                curBorrow: 0,
                harvestPerShare: 0,
                allocPoint: _allocpoint,
         totalMineRewards: 0,
         totalRecvHarvest: 0,
         totalRecvInterests: 0
            })
        );

        tokenOfPid[address(_token)] = lendPoolInfo.length - 1;

        spTokenOfPid[address(_spToken)] = tokenOfPid[address(_token)];

        totalAllocPoint = totalAllocPoint.add(_allocpoint);

        authSpTokenMap[_spToken] = true;

        borrowCompound.addBorrowPool(address(_token),_spToken);

        emit AddPoolEvent(msg.sender,address(_token),_allocpoint,_withUpdate);
    }
```

Listing 3.8: `ThemisLendCompound::addPool()`

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

Status The issue has been addressed by the following commit: `368c0d9`.

3.7 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

By design, the `ThemisLendCompound` contract is one of the main entries for interaction with users. In particular, one entry routine, i.e., `ThemisLendCompound::userLend()`, accepts user deposits of the supported `_pool.token` assets. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `ThemisLendCompound` contract. These asset-transferring routines will work well under the assumption that the vault's internal asset balances (specified by the `user.currTotalLend`) are always consistent with actual token balances maintained in individual ERC20 token contracts.

```
    function userLend(uint256 _pid, uint256 _amount) public authContractAccessChecker{
        require(_amount > 0, "lend invalid amount");

        CompoundLendPool storage _pool = lendPoolInfo[_pid];
        LendUserInfo storage user = lendUserInfos[msg.sender][_pid];

        uint256 _globalLendInterestShare = borrowCompound.getGlobalLendInterestShare(
            _pid);

        (uint256 _lendInterests,)  = pendingRedeemInterests(_pid,msg.sender);
        user.unRecvInterests = _lendInterests;
        user.lastLendInterestShare = _globalLendInterestShare;


        ThemisFinanceToken(_pool.spToken).mint(msg.sender, _amount);
        _pool.curSupply = _pool.curSupply.add(_amount);


        _updateCompound(_pid);

    user.currTotalLend = user.currTotalLend.add(_amount);
    user.userDli = user.userDli.add(_amount);


    _holderLendPoolIds[msg.sender].add(_pid);

    IERC20(_pool.token).safeTransferFrom(msg.sender, address(this), _amount);

        emit UserLend(msg.sender, _pid, _amount);
```

```
}
```

Listing 3.9: ThemisLendCompound::userLend()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer() or transferFrom(). (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as ThemisLendCompound::userLend(), may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of Themis V3 and affects protocol-wide operation and maintenance.

One possible mitigation is to regulate the set of ERC20 tokens that are permitted into Themis V3. In Themis V3, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., USDT) that may have control switches that can be dynamically exercised to suddenly become one.

Note other routines, i.e., ThemisEarlyFarming::userDeposit(), ThemisBorrowCompound::userReturn(), ThemisBorrowCompound::settlementBorrow() and ThemisAuction::doAuction(), share the same issue.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the transfer()/transferFrom() call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted USDT.

Status The issue has been confirmed by the team.

3.8 Trust Issue Of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the Themis V3 protocol, there is a privileged _governance account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the _governance account.

```
    function setThemisEarlyFarmingNFT(IThemisEarlyFarmingNFT _themisEarlyFarmingNFT)
        external onlyGovernance{
        themisEarlyFarmingNFT = _themisEarlyFarmingNFT;
    }
    ...
    function transferLendPoolReward(IERC20 _token,uint256 _amount) external
        onlyGovernance{
        IERC20(_token).safeTransfer(founder, _amount);
        emit TransferLendPoolRewardEvent(msg.sender,address(_token),founder,_amount);
    }
```

Listing 3.10: `ThemisEarlyFarming::setThemisEarlyFarmingNFT()&&transferLendPoolReward()`

```
    function setSpecial721BorrowRate(address special721,uint256 rate,string memory name)
         external onlyGovernance{
        uint256 beforeRate = special721Info[special721].rate;

        special721Info[special721].name = name;
        special721Info[special721].rate = rate;

        bool flag = true;
        for(uint i=0;i<special721Arr.length;i++){
            if(special721Arr[i] == special721){
                flag = false;
                break;
            }
        }
        if(flag){
            special721Arr[special721Arr.length] = special721;
        }

        emit SetSpecial721BorrowRateEvent(special721,msg.sender,beforeRate,rate);
    }
```

Listing 3.11: `ThemisBorrowCompound::setSpecial721BorrowRate()`

```
    function abortiveAuction(uint256 auctionId) external{
        require(_holderAuctioningIds.contains(auctionId),"This auction not exist.");
        (uint256 _auctionAmount,,,, bool _bidFlag,,) = _getCurrSaleInfo(auctionId,false)
            ;
        require(_auctionAmount == 0,"In time.");
        require(!_bidFlag,"already bid.");

        AuctionInfo storage _auctionInfo = auctionInfos[auctionId];
        _auctionInfo.state = 9;

        _holderAuctioningIds.remove(auctionId);
        _holderBidAuctionIds.remove(auctionId);
        IERC721(_auctionInfo.erc721Addr).transferFrom(address(this), funder,
            _auctionInfo.tokenId);

        emit AbortiveAuctionEvent(auctionId,funder);
```

```
    }
```

Listing 3.12: `ThemisAuction::abortiveAuction()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the `_governance` is not governed by a `DAO`-like structure. Note that a compromised `_governance` account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `Themis V3` design.

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 This issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the `Themis V3` design and implementation. `Themis V3` is a permissionless decentralized protocol that provides lending and borrowing services through smart contracts, which allows the users to lend assets to receive an interest rate and borrow assets with `UniswapV3 position` as collateral. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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.