SMART CONTRACT AUDIT REPORT

for

Venus Isolated Pool

Prepared By: Xiaomi Huang

PeckShield

August 25, 2023

Document Properties

Client	Venus
Title	Smart Contract Audit Report
Target	Venus Isolated Pool
Version	1.1
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.1	August 25, 2023	Xuxian Jiang	Final Release
1.1-rc	June 19, 2023	Luck Hu	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Isolated Pool` support in the `Venus` 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 Venus Isolated Pool

The `Venus Isolated Pool` separates the collaterals into independent lending environments so that lenders and traders can choose to participate based on their personal risk preferences. With lending pools of varying risk, `Venus` expands beyond the typical risk-conservative `Financial Primitive` customer base, and changes its brand narrative into one that can participate in the latest innovative protocol tokens. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Venus Isolated Pool

Item	Description
Name	Venus
Website	https://venus.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 25, 2023

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

- https://github.com/VenusProtocol/isolated-pools.git (f075e82)

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

- https://github.com/VenusProtocol/isolated-pools.git (9676970)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;

- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	3	■ ■ ■
Informational	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 1 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key Venus Isolated Pool Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Lack of auctionsPaused Check in Shortfall::restartAuction()	Business Logic	Fixed
PVE-002	Low	Improved Calculation of redeemAmount in VToken::_redeemFresh()	Coding Practices	Fixed
PVE-003	Medium	Conversion of Risk Fund to USD Value in Shortfall::_startAuction()	Business Logic	Fixed
PVE-004	High	Potential Denial-of-Service in Shortfall::placeBid()	Coding Practices	Fixed
PVE-005	Low	Trust Issue of Admin Keys	Security Features	Resolved

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 Lack of auctionsPaused Check in Shortfall::restartAuction()

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

Description

In `Venus Isolated Pool`, there is an auction contract `Shortfall` that is designed to auction off the risk fund accumulated in the `RiskFund` contract. The risk fund is auctioned in exchange for users paying off the pool's bad debt. An auction can be started by anyone once a pool's bad debt has reached a minimum value as long as the auctions are unpaused. Once the auctions are paused, it disables new auctions but allows current auctions proceed. While examining the logic to restart a stale auction, we notice it does not properly check if the auctions are currently paused or not.

In the following, we show the related code snippet of the `restartAuction()` routine, which is used to restart a stale auction. Specifically, it simply checks if the given auction has been started but is still in the stale state, i.e., no bidder in the first `waitForFirstBidder` blocks since the start block of the auction. However, it does not properly check if the auctions are currently paused or not. As a result, a stale auction can be restarted even when the auctions are paused, which is not expected.

```
    function restartAuction(address comptroller) external {
        Auction storage auction = auctions[comptroller];

        require(_isStarted(auction), "no on-going auction");
        require(_isStale(auction), "you need to wait for more time for first bidder");

        auction.status = AuctionStatus.ENDED;

        emit AuctionRestarted(comptroller, auction.startBlock);
        _startAuction(comptroller);
```

```
    }
```

Listing 3.1: Shortfall :: restartAuction ()

Recommendation Revisit the `restartAuction()` function and restart the auction only when the auctions are unpaused.

Status This issue has been fixed in the following commit: `6b1b3bb`.

3.2 Improved Calculation of redeemAmount in VToken::_redeemFresh()

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `VToken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

In order to facilitate the redeeming of `VTokens`, `Venus Isolated Pool` allows the redeemer to provide the amount of `VTokens` to redeem into the underlying tokens. While it properly converts between the `VToken` amount and the underlying amount, our analysis shows the conversions can be further improved.

To elaborate, we show below the code snippet of the `_redeemFresh()` routine which is called by users to redeem `VTokens`. When a user provides the amount of underlying he/she wants to receive from redeeming `VTokens`, the routine calculates the required amount of `VTokens` to be redeemed (line 886). Because of the rounding down calculation by default, it may can not redeem the desired amount of the underlying tokens from the calculated `VToken` amount. So, it takes the rounding up value as the final `VToken` amount to be redeemed (line 889). However, it comes to our attention that it does not properly recalculate the underlying amount based on the final `VToken` amount. As a result, the user may receive less underlying tokens than expected. Based on this, we suggest to recalculate the underlying amount according to the final `VToken` amount.

```
854     function _redeemFresh(
855         address redeemer,
856         uint256 redeemTokensIn,
857         uint256 redeemAmountIn
858     ) internal {
859         require(redeemTokensIn == 0  redeemAmountIn == 0, "one of redeemTokensIn or
            redeemAmountIn must be zero");
```

```
        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != _getBlockNumber()) {
            revert RedeemFreshnessCheck();
        }

        /* exchangeRate = invoke Exchange Rate Stored() */
        Exp memory exchangeRate = Exp({ mantissa: _exchangeRateStored() });

        uint256 redeemTokens;
        uint256 redeemAmount;
        /* If redeemTokensIn > 0: */
        if (redeemTokensIn > 0) {
            /*
             * We calculate the exchange rate and the amount of underlying to be redeemed:
             *  redeemTokens = redeemTokensIn
             *  redeemAmount = redeemTokensIn x exchangeRateCurrent
             */
            redeemTokens = redeemTokensIn;
            redeemAmount = mul_ScalarTruncate(exchangeRate, redeemTokensIn);
        } else {
            /*
             * We get the current exchange rate and calculate the amount to be redeemed:
             *  redeemTokens = redeemAmountIn / exchangeRate
             *  redeemAmount = redeemAmountIn
             */
            redeemTokens = div_(redeemAmountIn, exchangeRate);

            uint256 _redeemAmount = mul_(redeemTokens, exchangeRate);
            if (_redeemAmount != 0 && _redeemAmount != redeemAmountIn) redeemTokens++; // round up
            redeemAmount = redeemAmountIn;
        }

        // Revert if tokens is zero and amount is nonzero or token is nonzero and amount is zero
        if ((redeemTokens == 0 && redeemAmount > 0)  (redeemTokens != 0 && redeemAmount == 0)) {
            revert("redeemTokens or redeemAmount is zero");
        }
        ...
    }
```

Listing 3.2: `VToken::_redeemFresh()`

Recommendation Revisit the calculation of the `VToken` amount that is needed to receive the desired underlying amount in the `VToken::_redeemFresh()` routine, and recalculate the underlying amount based on the final `VToken` amount.

Status This issue has been fixed in the following commit: `6b1b3bb`.

3.3 Conversion of Risk Fund to USD Value in Shortfall::_startAuction()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Shortfall
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, in `Venus Isolated Pool`, `Shortfall` is an auction contract designed to auction off the risk fund accumulated in the `RiskFund` contract in exchange for the auction winner paying off the pool's bad debt. It designs two strategies that can be chosen to run the auction depending on the risk fund value and the bad debt value with a specific incentive. While examining the comparison of these two values, we notice they are compared in different units which shall be corrected.

To elaborate, we show below the code snippet of the `Shortfall::_startAuction()` routine which is used to start an auction to pay off the bad debt for the given `Comptroller`. Specifically, it loops all the pools in the `Comptroller` and counts all the bad debt in `USD` (lines 424 – 433), reads the risk fund balance from the `RiskFund` contract (line 438), calculates the incentivized bad debt value with the specific incentive (line 440), and compares the incentivized bad debt value with the risk fund balance (line 441). However, we notice that the risk fund balance is counted in the unit of the base asset (e.g., `USDT`), while the incentivized bad debt value is counted in `USD`. As a result, the comparison of the two values in different units may lead to unexpected result, possibly leading to the selection of an unexpected auction strategy.

Based on this, we suggest to convert the risk fund balance to its `USD` value before comparing it with the incentivized bad debt value.

```
414     function _startAuction(address comptroller) internal {
415         ...
416         VToken[] memory vTokens = _getAllMarkets(comptroller);
417         marketsCount = vTokens.length;
418         ResilientOracleInterface priceOracle = _getPriceOracle(comptroller);
419         uint256 poolBadDebt;
420
421         uint256[] memory marketsDebt = new uint256[](marketsCount);
422         auction.markets = new VToken[](marketsCount);
423
424         for (uint256 i; i < marketsCount; ++i) {
425             uint256 marketBadDebt = vTokens[i].badDebt();
426
```

```
            priceOracle.updatePrice(address(vTokens[i]));
            uint256 usdValue = (priceOracle.getUnderlyingPrice(address(vTokens[i])) *
                marketBadDebt) / EXP_SCALE;

            poolBadDebt = poolBadDebt + usdValue;
            auction.markets[i] = vTokens[i];
            auction.marketDebt[vTokens[i]] = marketBadDebt;
            marketsDebt[i] = marketBadDebt;
        }

        require(poolBadDebt >= minimumPoolBadDebt, "pool bad debt is too low");

        uint256 riskFundBalance = riskFund.poolReserves(comptroller);
        uint256 remainingRiskFundBalance = riskFundBalance;
        uint256 incentivizedRiskFundBalance = poolBadDebt + ((poolBadDebt * incentiveBps
            ) / MAX_BPS);
        if (incentivizedRiskFundBalance >= riskFundBalance) {
            auction.startBidBps =
                (MAX_BPS * MAX_BPS * remainingRiskFundBalance) /
                (poolBadDebt * (MAX_BPS + incentiveBps));
            remainingRiskFundBalance = 0;
            auction.auctionType = AuctionType.LARGE_POOL_DEBT;
        } else {
            uint256 maxSeizeableRiskFundBalance = incentivizedRiskFundBalance;

            remainingRiskFundBalance = remainingRiskFundBalance -
                maxSeizeableRiskFundBalance;
            auction.auctionType = AuctionType.LARGE_RISK_FUND;
            auction.startBidBps = MAX_BPS;
        }

        auction.seizedRiskFund = riskFundBalance - remainingRiskFundBalance;
        auction.startBlock = block.number;
        auction.status = AuctionStatus.STARTED;
        auction.highestBidder = address(0);

        emit AuctionStarted(...);
    }
```

Listing 3.3: Shortfall :: _startAuction()

Recommendation Properly convert the risk fund balance to its USD value before comparing it with the incentivized bad debt value.

Status This issue has been fixed in the following commit: 9676970.

3.4 Potential Denial-of-Service in Shortfall::placeBid()

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Shortfall`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

In `Venus Isolated Pool`, a user can bid for an ongoing auction by placing a higher bid than the current highest bidder. Before becoming the new highest bidder, it has to return the bid to the current highest bidder. While examining the returning of the bid to the highest bidder, we notice the risk of denial-of-service that may block new bidder to bid for the auction.

In the following, we show the code snippet of the `Shortfall::placeBid()` routine which is used to bid for an ongoing auction. Generally, if this is a valid auction, i.e., the auction has been started and is not stale, and the new bid is higher than the current highest bid, a new highest bidder is generated. The current highest bid will be returned back to the current highest bidder (line 204). However, it comes to our attention that if the `Erc20` token is an `ERC777`-like one which may have a callback function to the receiver contract, the current highest bidder, i.e., the transfer receiver, may revert the token transfer, hence the returning of the highest bid reverts and nobody can bid for this auction anymore. As a result, the current highest bidder can finally win the auction and get the risk fund.

Based on this, we suggest to properly handle the result of returning the highest bid and design a way for the old highest bidder to claim its returned bid if the returning of the bid can not be successful.

```
190     function placeBid(address comptroller, uint256 bidBps) external nonReentrant {
191         Auction storage auction = auctions[comptroller];

193         require(_isStarted(auction), "no on-going auction");
194         require(!_isStale(auction), "auction is stale, restart it");
195         require(bidBps <= MAX_BPS, "basis points cannot be more than 10000");
196         require(...);

198         uint256 marketsCount = auction.markets.length;
199         for (uint256 i; i < marketsCount; ++i) {
200             VToken vToken = VToken(address(auction.markets[i]));
201             IERC20Upgradeable erc20 = IERC20Upgradeable(address(vToken.underlying()));

203             if (auction.highestBidder != address(0)) {
204                 erc20.safeTransfer(auction.highestBidder, auction.bidAmount[auction.
                        markets[i]]);
```

```
            }
            uint256 balanceBefore = erc20.balanceOf(address(this));

            if (auction.auctionType == AuctionType.LARGE_POOL_DEBT) {
                uint256 currentBidAmount = ((auction.marketDebt[auction.markets[i]] *
                    bidBps) / MAX_BPS);
                erc20.safeTransferFrom(msg.sender, address(this), currentBidAmount);
            } else {
                erc20.safeTransferFrom(msg.sender, address(this), auction.marketDebt[
                    auction.markets[i]]);
            }

            uint256 balanceAfter = erc20.balanceOf(address(this));
            auction.bidAmount[auction.markets[i]] = balanceAfter - balanceBefore;
        }

        auction.highestBidder = msg.sender;
        auction.highestBidBps = bidBps;
        auction.highestBidBlock = block.number;

        emit BidPlaced(comptroller, auction.startBlock, bidBps, msg.sender);
    }
```

Listing 3.4: Shortfall :: placeBid()

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

Recommendation Take care of the result of returning the bid to the current highest bidder and design a way for the old highest bidder to claim its returned bid.

Status This issue has been fixed in the following commit: `a689850`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `Venus Isolated Pool` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the protocol-wide operations (e.g., set the price oracle). Our analysis shows that this privileged account need to be scrutinized. In the following, we show the representative functions potentially affected by the privileges of the `owner` account.

Firstly, the privileged functions in `Comptroller` allow the `owner` account to set the price oracle for the comptroller which provides prices for the underlying tokens in the comptroller, set the max loops limit to avoid `DOS`, etc.

```
    function setPriceOracle(ResilientOracleInterface newOracle) external onlyOwner {
        ensureNonzeroAddress(address(newOracle));

        ResilientOracleInterface oldOracle = oracle;
        oracle = newOracle;
        emit NewPriceOracle(oldOracle, newOracle);
    }

    function setMaxLoopsLimit(uint256 limit) external onlyOwner {
        _setMaxLoopsLimit(limit);
    }
```

Listing 3.5: Example Privileged Operations for `Comptroller`

Secondly, the privileged functions in `VToken` allow the `owner` account to set the protocol share reserve contract address which is used to receive the protocol reserves, set the short fall contract address which can recover the pool's bad debt.

```
    function setProtocolShareReserve(address payable protocolShareReserve_) external
        onlyOwner {
        _setProtocolShareReserve(protocolShareReserve_);
    }

    function setShortfallContract(address shortfall_) external onlyOwner {
        _setShortfallContract(shortfall_);
    }
```

Listing 3.6: Example Privileged Operations for `VToken`

At last, the privileged functions in `RewardsDistributor` allow the `owner` account to grant any amount of the reward token to any recipient, set the reward speed for the given contributor, etc.

```
    function grantRewardToken(address recipient, uint256 amount) external onlyOwner {
        uint256 amountLeft = _grantRewardToken(recipient, amount);
        require(amountLeft == 0, "insufficient rewardToken for grant");
        emit RewardTokenGranted(recipient, amount);
    }

    function setContributorRewardTokenSpeed(address contributor, uint256
        rewardTokenSpeed) external onlyOwner {
        // note that REWARD TOKEN speed could be set to 0 to halt liquidity rewards for
            a contributor
        updateContributorRewards(contributor);
        if (rewardTokenSpeed == 0) {
            // release storage
            delete lastContributorBlock[contributor];
        } else {
            lastContributorBlock[contributor] = getBlockNumber();
```

```
        }
        rewardTokenContributorSpeeds[contributor] = rewardTokenSpeed;

        emit ContributorRewardTokenSpeedUpdated(contributor, rewardTokenSpeed);
    }
```

Listing 3.7: Example Privileged Operations for `VToken`

We do not list all the privileged functions in the protocol. While the privilege assignment may be necessary and consistent with the protocol design, it could be worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised `owner` account would allow the attacker to modify the sensitive system parameter, which directly undermines the assumption of the `Venus Isolated Pool` design.

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

Status This issue has been resolved as the `owner` is managed by a governance timelock contract.

4 Conclusion

In this audit, we have analyzed the `Venus Isolated Pool` design and implementation. The `Venus Isolated Pool` separates the collaterals into independent lending environments and lenders and traders can choose to participate based on their personal risk preferences. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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

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