SMART CONTRACT AUDIT REPORT

for

Venus Isolated Pool

Prepared By: Xiaomi Huang

PeckShield

January 12, 2023

Document Properties

Client	Venus
Title	Smart Contract Audit Report
Target	Venus Isolated Pool
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 12, 2023	Xiaotao Wu	Final Release
1.0-rc	December 28, 2022	Xiaotao Wu	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 current `Venus` collateral pool is a common pool, which means all the assets are vulnerable to bankruptcies in a single asset. This is good for the capital efficiency of the included tokens, but requires a very conservative risk profile of chosen assets. More risky assets do not have the opportunity for lending because the net risk of the included assets cannot exceed the most conservative lender in the target market. 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. 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	January 12, 2023

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

- https://github.com/VenusProtocol/isolated-pools.git (41d131a)
- https://github.com/VenusProtocol/oracle.git (883e385)

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

- https://github.com/VenusProtocol/isolated-pools.git (d644aba)
- https://github.com/VenusProtocol/oracle.git (883e385)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

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

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

- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

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

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

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

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.

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.

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	2	■■
Medium	3	■■■
Low	3	■■■
Informational	2	■■
Total	10	

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, 3 low-severity vulnerabilities, and 2 informational suggestions.

Table 2.1: Key Venus Isolated Pool Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Function Argument Passed in VToken::_liquidateBorrowFresh()	Business Logic	Fixed
PVE-002	Low	Possible Front-Running For Unintended Payment In repayBorrowBehalf()	Time and State	Fixed
PVE-003	High	Incorrect Liquidation Target in Comptroller::liquidateAccount()	Business Logic	Fixed
PVE-004	Low	Missed Sanity Checks in Comptroller::liquidateAccount()	Coding Practices	Fixed
PVE-005	Informational	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-006	Low	Improved Implementation Logic in Shortfall::closeAuction()	Business Logic	Fixed
PVE-007	Medium	Revisited Implementation Logic in ChainlinkOracle	Business Logic	Fixed
PVE-008	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed
PVE-009	High	Incorrect Bad Debt Recovered Amount in Shortfall::closeAuction()	Business Logic	Fixed
PVE-010	Medium	Incorrect startBidBps Initialization in Shortfall::startAuction()	Business Logic	Fixed

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 Incorrect Function Argument Passed in VToken::_liquidateBorrowFresh()

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `VToken`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

In `Venus Isolated Pool`, a borrower can be liquidated if this borrower's collateral is under-collateralized or is less than a predefined threshold. The `VToken` contract provides a helper `_liquidateBorrowFresh()` routine to assist the liquidation to a given borrower. While reviewing its logic, we notice the current implementation is not correct.

In the following, we show the related code snippet of the `_liquidateBorrowFresh()` routine. Specifically, while reviewing the `comptroller.preLiquidateHook()` logic, we notice the `liquidator` is wrongly passed as input for the `vTokenBorrowed` argument of the `preLiquidateHook()` function (line 893). The correct input for the `vTokenBorrowed` argument should be the address of the `VToken` contract, i.e., `address(this)`.

```
884     function _liquidateBorrowFresh(
885         address liquidator,
886         address borrower,
887         uint256 repayAmount,
888         VTokenInterface vTokenCollateral,
889         bool skipLiquidityCheck
890     ) internal {
891         /* Fail if liquidate not allowed */
892         comptroller.preLiquidateHook(
893             liquidator,
894             address(vTokenCollateral),
895             liquidator,
```

```
            borrower,
            repayAmount,
            skipLiquidityCheck
        );

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

        ...
    }
```

Listing 3.1: VToken::_liquidateBorrowFresh()

Recommendation Set `address(this)` as the `vTokenBorrowed` input argument of the `preLiquidateHook()` function.

Status This issue has been fixed in the following commit: `75d02da`.

3.2 Possible Front-Running For Unintended Payment In repayBorrowBehalf()

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `VToken`
- Category: Time and State [7]
- CWE subcategory: CWE-663 [3]

Description

The `Venus Isolated Pool` is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying and borrowing users, i.e., `mint()/redeem()` and `borrow()/repayBorrow()`. In the following, we examine one specific functionality, i.e., `repayBorrow()`.

To elaborate, we show below the core routine `_repayBorrowFresh()` that actually implements the main logic behind the `repayBorrow()` routine. This routine allows for repaying partial or full current borrowing balance. It is interesting to note that the `Venus` protocol supports the payment on behalf of another borrowing user (via `repayBorrowBehalf()`). And the `_repayBorrowFresh()` routine supports the corner case when the given amount is larger than the current borrowing balance. In this corner case, the protocol assumes the intention for a full repayment.

```
    function _repayBorrowFresh(
```

```
        address payer,
        address borrower,
        uint256 repayAmount
    ) internal returns (uint256) {
        /* Fail if repayBorrow not allowed */
        comptroller.preRepayHook(address(this), payer, borrower, repayAmount);

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

        /* We fetch the amount the borrower owes, with accumulated interest */
        uint256 accountBorrowsPrev = _borrowBalanceStored(borrower);

        /* If repayAmount == -1, repayAmount = accountBorrows */
        uint256 repayAmountFinal = repayAmount == type(uint256).max ? accountBorrowsPrev
             : repayAmount;

        /////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
         * We call _doTransferIn for the payer and the repayAmount
         *  Note: The vToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the vToken holds an additional repayAmount of cash.
         *  _doTransferIn reverts if anything goes wrong, since we can't be sure if side
              effects occurred.
         *   it returns the amount actually transferred, in case of a fee.
         */
        uint256 actualRepayAmount = _doTransferIn(payer, repayAmountFinal);

        /*
         * We calculate the new borrower and total borrow balances, failing on underflow
             :
         *  accountBorrowsNew = accountBorrows - actualRepayAmount
         *  totalBorrowsNew = totalBorrows - actualRepayAmount
         */
        uint256 accountBorrowsNew = accountBorrowsPrev - actualRepayAmount;
        uint256 totalBorrowsNew = totalBorrows - actualRepayAmount;

        /* We write the previously calculated values into storage */
        accountBorrows[borrower].principal = accountBorrowsNew;
        accountBorrows[borrower].interestIndex = borrowIndex;
        totalBorrows = totalBorrowsNew;

        /* We emit a RepayBorrow event */
        emit RepayBorrow(payer, borrower, actualRepayAmount, accountBorrowsNew,
            totalBorrowsNew);

        return actualRepayAmount;
```

```
    }
```

Listing 3.2: `VToken::_repayBorrowFresh()`

This is a reasonable assumption, but our analysis shows this assumption may be taken advantage of to launch a front-running `borrow()` operation, resulting in a higher borrowing balance for repayment. To avoid this situation, it is suggested to disallow the repayment amount of `type(uint256).max` to imply the full repayment. In fact, it is always suggested to use the exact payment amount in the `repayBorrowBehalf()` case.

Recommendation Revisit the generous assumption of using repayment amount of `type(uint256).max` as the indication of full repayment.

Status This issue has been fixed in the following commit: `6bcda4f`.

3.3 Incorrect Liquidation Target in Comptroller::liquidateAccount()

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Comptroller`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, in `Venus Isolated Pool`, a borrower can be liquidated if this borrower's collateral is under-collateralized or is less than a predefined threshold. The `Comptroller` contract provides an external `liquidateAccount()` function for liquidators to liquidate borrowers whose collateral are less than the predefined threshold. Our analysis with the `liquidateAccount()` function shows its current implementation is not correct.

To elaborate, we show below the code snippet of the `liquidateAccount()` routine. This routine implements a rather straightforward logic in firstly validating the collateral of a borrower is less than the predefined threshold and the liquidation is profitable, then the liquidator repays all the borrows belonging to this borrower and seizes all the collaterals from this borrower. However, it comes to our attention that the liquidation target market is not correctly used (line 700). Specifically, the `forceLiquidateBorrow()` function of the `order.vTokenBorrowed` should be invoked, instead of the current invoking of `order.vTokenCollateral.forceLiquidateBorrow()`.

```
677     function liquidateAccount(address borrower, LiquidationOrder[] calldata orders)
            external {
```

```
        // We will accrue interest and update the oracle prices later during the liquidation

        AccountLiquiditySnapshot memory snapshot = _getCurrentLiquiditySnapshot(borrower, _getLiquidationThreshold);

        if (snapshot.totalCollateral > minLiquidatableCollateral) {
            // You should use the regular vToken.liquidateBorrow(...) call
            revert CollateralExceedsThreshold(minLiquidatableCollateral, snapshot.totalCollateral);
        }

        uint256 collateralToSeize = mul_ScalarTruncate(
            Exp({ mantissa: liquidationIncentiveMantissa }),
            snapshot.borrows
        );
        if (collateralToSeize >= snapshot.totalCollateral) {
            // There is not enough collateral to seize. Use healBorrow to repay some part of the borrow
            // and record bad debt.
            revert InsufficientCollateral(collateralToSeize, snapshot.totalCollateral);
        }

        uint256 ordersCount = orders.length;
        for (uint256 i; i < ordersCount; ++i) {
            LiquidationOrder calldata order = orders[i];
            order.vTokenCollateral.forceLiquidateBorrow(
                msg.sender,
                borrower,
                order.repayAmount,
                order.vTokenCollateral,
                true
            );
        }

        VToken[] memory markets = accountAssets[borrower];
        uint256 marketsCount = markets.length;
        for (uint256 i; i < marketsCount; ++i) {
            (, uint256 borrowBalance, ) = _safeGetAccountSnapshot(markets[i], borrower);
            require(borrowBalance == 0, "Nonzero borrow balance after liquidation");
        }
    }
```

Listing 3.3: Comptroller::liquidateAccount()

Recommendation Invoke the `forceLiquidateBorrow()` function from the correct market.

Status This issue has been fixed in the following commit: `4e9070f`.

3.4 Missed Sanity Checks in Comptroller::liquidateAccount()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [2]

Description

As mentioned earlier, the `liquidateAccount()` function of the `Comptroller` contract allows for liquidators to liquidate borrowers whose collateral are less than the predefined threshold. While reviewing its logic, we notice the current implementation fails to check the given argument in `orders`.

To elaborate, we show below the code snippet of the `liquidateAccount()` function. Since the input argument `orders` is determined by the function caller and thus the `forceLiquidateBorrow()` function invoked from the `order.vTokenCollateral` contract may be trustless. One possible scenario is that a malicious actor can reenter the `liquidateAccount()` function if the input argument `order.vTokenCollateral` is not a real `Venus Isolated Pool` market address (line 700).

```
677     function liquidateAccount(address borrower, LiquidationOrder[] calldata orders) external {
678         // We will accrue interest and update the oracle prices later during the liquidation
679 
680         AccountLiquiditySnapshot memory snapshot = _getCurrentLiquiditySnapshot(borrower, _getLiquidationThreshold);
681 
682         if (snapshot.totalCollateral > minLiquidatableCollateral) {
683             // You should use the regular vToken.liquidateBorrow(...) call
684             revert CollateralExceedsThreshold(minLiquidatableCollateral, snapshot.totalCollateral);
685         }
686 
687         uint256 collateralToSeize = mul_ScalarTruncate(
688             Exp({ mantissa: liquidationIncentiveMantissa }),
689             snapshot.borrows
690         );
691         if (collateralToSeize >= snapshot.totalCollateral) {
692             // There is not enough collateral to seize. Use healBorrow to repay some part of the borrow
693             // and record bad debt.
694             revert InsufficientCollateral(collateralToSeize, snapshot.totalCollateral);
695         }
696 
697         uint256 ordersCount = orders.length;
698         for (uint256 i; i < ordersCount; ++i) {
699             LiquidationOrder calldata order = orders[i];
```

```
            order.vTokenCollateral.forceLiquidateBorrow(
                msg.sender,
                borrower,
                order.repayAmount,
                order.vTokenCollateral,
                true
            );
        }

        VToken[] memory markets = accountAssets[borrower];
        uint256 marketsCount = markets.length;
        for (uint256 i; i < marketsCount; ++i) {
            (, uint256 borrowBalance, ) = _safeGetAccountSnapshot(markets[i], borrower);
            require(borrowBalance == 0, "Nonzero borrow balance after liquidation");
        }
    }
```

Listing 3.4: Comptroller::liquidateAccount()

Note similar issue also exists in the `ReserveHelpers::updateAssetsState()` routine and in the `PoolRegistry::addMarket()` routine.

Recommendation Add necessary sanity checks for the above mentioned functions.

Status This issue has been confirmed. This issue has been fixed in the following commit: `55a1eac`.

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Shortfall`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.5: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In current implementation, if we examine the `Shortfall::closeAuction()` routine that is designed to close an on-going auction. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 220).

```
182     function closeAuction(address comptroller) external nonReentrant {
183         Auction storage auction = auctions[comptroller];
184
185         require(auction.startBlock != 0 && auction.status == AuctionStatus.STARTED, "no
                on-going auction");
186         require(
187             block.number > auction.highestBidBlock + nextBidderBlockLimit && auction.
                    highestBidder != address(0),
188             "waiting for next bidder. cannot close auction"
189         );
190
191         uint256 marketsCount = auction.markets.length;
192         uint256[] memory marketsDebt = new uint256[](marketsCount);
193
```

```
        auction.status = AuctionStatus.ENDED;

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

            if (auction.auctionType == AuctionType.LARGE_POOL_DEBT) {
                uint256 bidAmount = ((auction.marketDebt[auction.markets[i]] * auction.
                    highestBidBps) / MAX_BPS);
                erc20.safeTransfer(address(auction.markets[i]), bidAmount);
                marketsDebt[i] = bidAmount;
            } else {
                erc20.safeTransfer(address(auction.markets[i]), auction.marketDebt[
                    auction.markets[i]]);
                marketsDebt[i] = auction.marketDebt[auction.markets[i]];
            }

            auction.markets[i].badDebtRecovered(auction.marketDebt[auction.markets[i]]);
        }

        uint256 riskFundBidAmount = auction.seizedRiskFund;

        if (auction.auctionType == AuctionType.LARGE_POOL_DEBT) {
            riskFund.transferReserveForAuction(comptroller, riskFundBidAmount);
            BUSD.safeTransfer(auction.highestBidder, riskFundBidAmount);
        } else {
            riskFundBidAmount = (auction.seizedRiskFund * auction.highestBidBps) /
                MAX_BPS;
            riskFund.transferReserveForAuction(comptroller, riskFundBidAmount);
            BUSD.transfer(auction.highestBidder, riskFundBidAmount);
        }

        emit AuctionClosed(
            comptroller,
            auction.highestBidder,
            auction.highestBidBps,
            riskFundBidAmount,
            auction.markets,
            marketsDebt
        );
    }
```

Listing 3.6: `Shortfall::closeAuction()`

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

Status This issue has been fixed in the following commit: `f4129a4`.

3.6 Improved Implementation Logic in Shortfall::closeAuction()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Shortfall`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

In `Venus Isolated Pool`, the bad debt can be recovered via auction. A user can repay the bad debt by bidding and receive the risk fund as return. Once a new bid has been placed, this on-going auction can be closed if there is no bidder who offers a higher price in the next 10 blocks. While examining the `closeAuction()` routine of the `Shortfall` contract, we notice the current implementation logic can be improved.

To elaborate, we show below the related code snippet. It comes to our attention that the `BUSD` asset a user will get after repaying the bad debt comes from the `RiskFund` contract (lines 214-221). In `RiskFund`, the pool assets will be swapped to `convertibleBaseAsset` via `PancakeswapV2` and transferred to the `Shortfall` contract for auction. The storage variable `convertibleBaseAsset` of the `RiskFund` contract can be changed by the privileged `owner` account while the storage variable `BUSD` of the `RiskFund` contract is unchangeable once initialized. Thus the execution of the `closeAuction()` routine will revert if `BUSD != convertibleBaseAsset`.

```
182     function closeAuction(address comptroller) external nonReentrant {
183         Auction storage auction = auctions[comptroller];
184 
185         require(auction.startBlock != 0 && auction.status == AuctionStatus.STARTED, "no
            on-going auction");
186         require(
187             block.number > auction.highestBidBlock + nextBidderBlockLimit && auction.
                highestBidder != address(0),
188             "waiting for next bidder. cannot close auction"
189         );
190 
191         uint256 marketsCount = auction.markets.length;
192         uint256[] memory marketsDebt = new uint256[](marketsCount);
193 
194         auction.status = AuctionStatus.ENDED;
195 
196         for (uint256 i; i < marketsCount; ++i) {
197             VToken vToken = VToken(address(auction.markets[i]));
198             IERC20Upgradeable erc20 = IERC20Upgradeable(address(vToken.underlying()));
199 
200             if (auction.auctionType == AuctionType.LARGE_POOL_DEBT) {
```

```
                uint256 bidAmount = ((auction.marketDebt[auction.markets[i]] * auction.
                    highestBidBps) / MAX_BPS);
                erc20.safeTransfer(address(auction.markets[i]), bidAmount);
                marketsDebt[i] = bidAmount;
            } else {
                erc20.safeTransfer(address(auction.markets[i]), auction.marketDebt[
                    auction.markets[i]]);
                marketsDebt[i] = auction.marketDebt[auction.markets[i]];
            }

            auction.markets[i].badDebtRecovered(auction.marketDebt[auction.markets[i]]);
        }

        uint256 riskFundBidAmount = auction.seizedRiskFund;

        if (auction.auctionType == AuctionType.LARGE_POOL_DEBT) {
            riskFund.transferReserveForAuction(comptroller, riskFundBidAmount);
            BUSD.safeTransfer(auction.highestBidder, riskFundBidAmount);
        } else {
            riskFundBidAmount = (auction.seizedRiskFund * auction.highestBidBps) /
                MAX_BPS;
            riskFund.transferReserveForAuction(comptroller, riskFundBidAmount);
            BUSD.transfer(auction.highestBidder, riskFundBidAmount);
        }

        emit AuctionClosed(
            comptroller,
            auction.highestBidder,
            auction.highestBidBps,
            riskFundBidAmount,
            auction.markets,
            marketsDebt
        );
    }
```

Listing 3.7: `Shortfall::closeAuction()`

Recommendation Implement a privileged function to allow the `owner` account to change the storage variable `BUSD` of the `RiskFund` contract.

Status This issue has been fixed in the following commit: `c178a69`.

3.7 Revisited Implementation Logic in ChainlinkOracle

- ID: PVE-007
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `Multiple contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The `ChainlinkOracle` contract provides a public `getUnderlyingPrice()` function to get the `Chainlink` price of underlying asset of the input `vToken`. While reviewing its logic, we notice the current implementation needs to be revisited.

In the following, we show the related code snippet of the `getUnderlyingPrice()` routine. When the input `vToken` is `vBNB`, the internal `_getChainlinkPrice()` function will be called directly to obtain the price. However, the `vBNB` token doesn't have the `underlying` method. Thus the invoking of the `_getChainlinkPrice()` function will revert (line 99).

```
    function getUnderlyingPrice(address vToken) public view override returns (uint256) {
        string memory symbol = VBep20Interface(vToken).symbol();
        // VBNB token doesn't have `underlying` method, so it has to skip `_getUnderlyingPriceInternal
        // method and directly goes into `_getChainlinkPrice`
        if (_compareStrings(symbol, "vBNB")) {
            return _getChainlinkPrice(vToken);
            // VAI price is constantly 1 at the moment, but not guarantee in the future
        } else if (_compareStrings(symbol, "VAI")) {
            return VAI_VALUE;
            // @TODO: This is some history code, keep it here in case of messing up
        } else {
            return _getUnderlyingPriceInternal(VBep20Interface(vToken));
        }
    }
```

Listing 3.8: ChainlinkOracle::getUnderlyingPrice()

```
96      function _getChainlinkPrice(
97          address vToken
98      ) internal view notNullAddress(tokenConfigs[VBep20Interface(vToken).underlying()].asset) returns (uint256) {
99          address asset = VBep20Interface(vToken).underlying();
100
101         TokenConfig storage tokenConfig = tokenConfigs[asset];
102         AggregatorV2V3Interface feed = AggregatorV2V3Interface(tokenConfig.feed);
103
104         // note: maxStalePeriod cannot be 0
105         uint256 maxStalePeriod = tokenConfig.maxStalePeriod;
```

```
        // Chainlink USD-denominated feeds store answers at 8 decimals, mostly
        uint256 decimalDelta = uint256(18) - feed.decimals();

        (, int256 answer, , uint256 updatedAt, ) = feed.latestRoundData();
        require(answer > 0, "chainlink price must be positive");

        require(block.timestamp > updatedAt, "updatedAt exceeds block time");
        uint256 deltaTime = block.timestamp - updatedAt;
        require(deltaTime <= maxStalePeriod, "chainlink price expired");

        return uint256(answer) * (10 ** decimalDelta);
    }
```

Listing 3.9: ChainlinkOracle::_getChainlinkPrice()

Note similar issue also exists in the `getUnderlyingPrice()` routine of the `BinanceOracle/PythOracle/TwapOracle` contracts.

Recommendation Revisit the implementation of the above-mentioned routines.

Status This issue has been fixed.

3.8 Inconsistency Between Document and Implementation

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ResilientOracle`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

There is a misleading comment description in the `ResilientOracle` contract, which brings unnecessary confusion to understand and maintain the contract implementation. Specifically, the comment is in the description of the `setTokenConfig()` routine. Specifically, it has been documented that an asset's token configs must have not be added before when setting token configs for this asset.

However, the current implementation shows that the privileged `owner` account can reset the token configs for an asset even if this asset's configs have been added before.

```
    /**
     * @notice Set single token configs, vToken MUST HAVE NOT be added before, and main
         oracle MUST NOT be zero address
     * @param tokenConfig token config struct
     */
    function setTokenConfig(
```

```
        TokenConfig memory tokenConfig
    ) public onlyOwner notNullAddress(tokenConfig.asset) notNullAddress(tokenConfig.
        oracles[uint256(OracleRole.MAIN)]) {
        tokenConfigs[tokenConfig.asset] = tokenConfig;
        emit TokenConfigAdded(
            tokenConfig.asset,
            tokenConfig.oracles[uint256(OracleRole.MAIN)],
            tokenConfig.oracles[uint256(OracleRole.PIVOT)],
            tokenConfig.oracles[uint256(OracleRole.FALLBACK)]
        );
    }
```

Listing 3.10: `ResilientOracle::setTokenConfig()`

Recommendation Ensure the consistency between documentation and implementation.

Status This issue has been fixed.

3.9 Incorrect Bad Debt Recovered Amount in Shortfall::closeAuction()

- ID: PVE-009
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Shortfall`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.6, in `Venus Isolated Pool`, the bad debt can be recovered via auction. A user can repay the bad debt by bidding and receive the risk fund as return. Once a new bid has been placed, this on-going auction can be closed if there is no bidder with a higher price in the next 10 blocks. While examining the `closeAuction()` routine of the `Shortfall` contract, we notice the recovered bad debt amount is not correct if the auction type is `LARGE_POOL_DEBT`.

To elaborate, we show below the code snippet of the `closeAuction()` routine. It comes to our attention that if the auction type is `LARGE_POOL_DEBT`, the bidder needs to transfer `((auction.marketDebt[auction.markets[i]] * auction.highestBidBps)/ MAX_BPS)` amount of underlying asset to the auction market (line 202). Thus the recovered bad debt should also be `((auction.marketDebt[auction.markets[i]] * auction.highestBidBps)/ MAX_BPS)`, instead of current `auction.marketDebt[auction.markets[i]]` (line 209).

```
182    function closeAuction(address comptroller) external nonReentrant {
183        Auction storage auction = auctions[comptroller];
```

```

        require(auction.startBlock != 0 && auction.status == AuctionStatus.STARTED, "no on-going auction");
        require(
            block.number > auction.highestBidBlock + nextBidderBlockLimit && auction.highestBidder != address(0),
            "waiting for next bidder. cannot close auction"
        );

        uint256 marketsCount = auction.markets.length;
        uint256[] memory marketsDebt = new uint256[](marketsCount);

        auction.status = AuctionStatus.ENDED;

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

            if (auction.auctionType == AuctionType.LARGE_POOL_DEBT) {
                uint256 bidAmount = ((auction.marketDebt[auction.markets[i]] * auction.highestBidBps) / MAX_BPS);
                erc20.safeTransfer(address(auction.markets[i]), bidAmount);
                marketsDebt[i] = bidAmount;
            } else {
                erc20.safeTransfer(address(auction.markets[i]), auction.marketDebt[auction.markets[i]]);
                marketsDebt[i] = auction.marketDebt[auction.markets[i]];
            }

            auction.markets[i].badDebtRecovered(auction.marketDebt[auction.markets[i]]);
        }

        uint256 riskFundBidAmount = auction.seizedRiskFund;

        if (auction.auctionType == AuctionType.LARGE_POOL_DEBT) {
            riskFund.transferReserveForAuction(comptroller, riskFundBidAmount);
            BUSD.safeTransfer(auction.highestBidder, riskFundBidAmount);
        } else {
            riskFundBidAmount = (auction.seizedRiskFund * auction.highestBidBps) / MAX_BPS;
            riskFund.transferReserveForAuction(comptroller, riskFundBidAmount);
            BUSD.transfer(auction.highestBidder, riskFundBidAmount);
        }

        emit AuctionClosed(
            comptroller,
            auction.highestBidder,
            auction.highestBidBps,
            riskFundBidAmount,
            auction.markets,
            marketsDebt
        );
```

```
231     }
```

Listing 3.11: `Shortfall::closeAuction()`

Recommendation Use the local variable `marketsDebt[i]` as the input argument of the `badDebtRecovered()` function (line 209).

Status This issue has been fixed in the following commit: `d102243`.

3.10 Incorrect startBidBps Initialization in Shortfall::startAuction()

- ID: PVE-010
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `Shortfall`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

In `Venus Isolated Pool`, the privileged `owner` account can start an auction for a specified `comptroller`'s bad debts. If the `startAuction()` function is invoked by the `owner`, the auction related parameters of the specified `comptroller` will be initialized. While examining this `startAuction()` routine of the `Shortfall` contract, we notice one auction-related parameter is not correctly assigned.

To elaborate, we show below the related code snippet. Specifically, if the auction type is `LARGE_POOL_DEBT`, the value assinged to the state variable `auction.startBidBps` should be `(MAX_BPS * MAX_BPS * remainingRiskFundBalance)/ poolBadDebt / (MAX_BPS + incentiveBps)`, instead of current `((MAX_BPS - incentiveBps)* remainingRiskFundBalance)/ poolBadDebt` (line 329).

```
285     function startAuction(address comptroller) public onlyOwner {
286         Auction storage auction = auctions[comptroller];
287         require(
288             (auction.startBlock == 0 && auction.status == AuctionStatus.NOT_STARTED)
289                 auction.status == AuctionStatus.ENDED,
290             "auction is on-going"
291         );
292
293         uint256 marketsCount = auction.markets.length;
294         for (uint256 i; i < marketsCount; ++i) {
295             VToken vToken = auction.markets[i];
296             auction.marketDebt[vToken] = 0;
297             auction.highestBidBps = 0;
298             auction.highestBidBlock = 0;
299         }
```

```
        delete auction.markets;

        VToken[] memory vTokens = _getAllMarkets(comptroller);
        marketsCount = vTokens.length;
        PriceOracle priceOracle = _getPriceOracle(comptroller);
        uint256 poolBadDebt;

        uint256[] memory marketsDebt = new uint256[](marketsCount);
        auction.markets = new VToken[](marketsCount);

        for (uint256 i; i < marketsCount; ++i) {
            uint256 marketBadDebt = vTokens[i].badDebt();

            priceOracle.updatePrice(address(vTokens[i]));
            uint256 usdValue = (priceOracle.getUnderlyingPrice(address(vTokens[i])) *
                marketBadDebt) / 1e18;

            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.getPoolReserve(comptroller);
        uint256 remainingRiskFundBalance = riskFundBalance;
        uint256 incentivizedRiskFundBalance = poolBadDebt + ((poolBadDebt * incentiveBps
            ) / MAX_BPS);
        if (incentivizedRiskFundBalance >= riskFundBalance) {
            auction.startBidBps = ((MAX_BPS - incentiveBps) * remainingRiskFundBalance)
                / poolBadDebt;
            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(
            comptroller,
            auction.startBlock,
```

```
            auction.auctionType,
            auction.markets,
            marketsDebt,
            auction.seizedRiskFund,
            auction.startBidBps
        );
    }
```

Listing 3.12: `Shortfall::startAuction()`

Recommendation Assign the correct value to the state variable `auction.startBidBps` when the auction type is `LARGE_POOL_DEBT`.

Status This issue has been fixed in the following commit: `f88230d`.

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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

[7] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.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.