



SMART CONTRACT AUDIT REPORT

for

Automatic Income Allocation (Venus)



Prepared By: Xiaomi Huang

PeckShield
August 12, 2023

Document Properties

Client	Venus
Title	Smart Contract Audit Report
Target	Venus Automatic Income Allocation
Version	1.0
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
1.0	August 12, 2023	Xuxian Jiang	Final Release
1.0-rc	August 5, 2023	Xuxian Jiang	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	4
1.1	About Venus	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Improved Enum Additions in ErrorReporter::FailureInfo	11
3.2	Missing Caller Validation in VBNBAdmin::reduceReserves()	12
3.3	Incorrect Borrow Cap Enforcement in isolated-pools	13
3.4	Removal of Unused Code	15
3.5	Accommodation of Non-ERC20-Compliant Tokens	16
3.6	Inaccurate Transfer Event in isolated-pools/VToken	19
4	Conclusion	21
	References	22

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the Automatic Income Allocation 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

The Venus protocol enables a complete algorithmic money market protocol on BNB Smart Chain. Venus enables users to utilize their cryptocurrencies by supplying collateral to the protocol that may be borrowed by pledging over-collateralized cryptocurrencies. The audited Automatic Income Allocation support allows for streamlined allocation and distribution of protocol income, either spread of liquidation. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Venus Automatic Income Allocation

Item	Description
Name	Venus
Website	https://venus.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 12, 2023

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

- <https://github.com/VenusProtocol/venus-protocol/pull/262>

- <https://github.com/VenusProtocol/venus-protocol/pull/289>
- <https://github.com/VenusProtocol/isolated-pools/pull/207>
- <https://github.com/VenusProtocol/protocol-reserve.git> (a49b870)

1.2 About PeckShield

PeckShield Inc. [8] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- 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) [6], 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 `Automatic Income Allocation` support in `Venus`. 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	0	
Medium	1	
Low	4	
Informational	1	
Total	6	

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 medium-severity vulnerability, 4 low-severity vulnerabilities, and 1 informational suggestion.

Table 2.1: Key Venus Automatic Income Allocation Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Enum Additions in ErrorReporter::FailureInfo	Coding Practices	Resolved
PVE-002	Low	Missing Caller Validation in VBNBAdmin::reduceReserves()	Coding Practices	Resolved
PVE-003	Medium	Incorrect Borrow Cap Enforcement in isolated-pools	Business Logic	Resolved
PVE-004	Informational	Unused State/Code Removal in ProtocolShareReserve	Coding Practices	Resolved
PVE-005	Low	Accommodating Non-ERC20-Compliant Tokens in RiskFund	Coding Practices	Resolved
PVE-006	Low	Inaccurate Transfer Event in isolated-pools/VToken	Coding Practices	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 Improved Enum Additions in ErrorReporter::FailureInfo

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ErrorReporter
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

In Venus, there is a systematic mapping between error code and the associated root cause. And the systematic mapping is maintained in the `ErrorReporter` contract. While examining the PR262, we notice the addition of three new error code. However, the addition is performed by not appending to the end of current error code list, instead by inserting into the middle of the error code list.

In the following, we show the related changes of the defined `enum` list, i.e., `FailureInfo`. We notice the addition of three new error code, `SET_ACCESS_CONTROL_OWNER_CHECK`, `SET_REDUCE_RESERVES_BLOCK_DELTA_OWNER_CHECK`, and `SET_PROTOCOL_SHARE_RESERVES_OWNER_CHECK`. While it does not affect the normal functionality, it may bring unnecessary confusion to the diagnosis of such errors in case they indeed occur.

```

168 @@ -168,6 +168,7 @@
169     SET_COLLATERAL_FACTOR_OWNER_CHECK ,
170     SET_COLLATERAL_FACTOR_VALIDATION ,
171     SET_COMPTROLLER_OWNER_CHECK ,
172 +   SET_ACCESS_CONTROL_OWNER_CHECK ,
173     SET_INTEREST_RATE_MODEL_ACCRUE_INTEREST_FAILED ,
174     SET_INTEREST_RATE_MODEL_FRESH_CHECK ,
175     SET_INTEREST_RATE_MODEL_OWNER_CHECK ,
176 @@ -178,6 +179,8 @@
177     SET_RESERVE_FACTOR_ADMIN_CHECK ,
178     SET_RESERVE_FACTOR_FRESH_CHECK ,
179     SET_RESERVE_FACTOR_BOUNDS_CHECK ,
180 +   SET_REDUCE_RESERVES_BLOCK_DELTA_OWNER_CHECK ,

```

```

181 +     SET_PROTOCOL_SHARE_RESERVES_OWNER_CHECK ,
182     TRANSFER_COMPTROLLER_REJECTION ,
183     TRANSFER_NOT_ALLOWED ,
184     TRANSFER_NOT_ENOUGH ,

```

Listing 3.1: `ErrorReporter::enum` `FailureInfo`

Recommendation Revisit the above `enum` so that new additions are always append to the end of current list.

Status This issue has been fixed in the following commit: `a575a70`.

3.2 Missing Caller Validation in `VBNBAdmin::reduceReserves()`

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `VBNBAdmin`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

In order to facilitate the automatic income collection and distribution, the `PR289` realizes the non-upgradeability of current `VBNB` market and proposes a new `VBNBAdmin` contract. This contract acts as the new admin of current `VBNB` market so that it is authorized to collect the revenue and distribute to the intended recipients. While examining the current revenue collection, we notice the reserve collection can be performed by anyone, which may not be consistent with other `VBep20` markets.

To elaborate, we show below the code snippet of the `reduceReserves()` routine. It comes to our attention that this routine may be called by anyone to collect the revenue. In comparison, other `VBep20` markets have the `_reduceReserves()` routine that is guarded with the access control module, i.e., `ensureAllowed("_reduceReserves(uint256)")` (line 249). For consistency, we suggest to use the same name `_reduceReserves()` with the same access control protection.

```

52     function reduceReserves(uint reduceAmount) external nonReentrant {
53         require(vBNB._reduceReserves(reduceAmount) == 0, "reduceReserves failed");
54         _wrapBNB();
55
56         uint256 balance = WBNB.balanceOf(address(this));
57         WBNB.safeTransfer(address(protocolShareReserve), balance);
58         protocolShareReserve.updateAssetsState(
59             vBNB.comptroller(),
60             address(WBNB),
61             IProtocolShareReserve.IncomeType.SPREAD
62         );

```

```

64     emit ReservesReduced(reduceAmount);
65 }

```

Listing 3.2: VBNBAdmin::reduceReserves()

```

248 function _reduceReserves(uint reduceAmount_) external nonReentrant returns (uint) {
249     ensureAllowed("_reduceReserves(uint256)");
250     uint error = accrueInterest();
251     if (error != uint(Error.NO_ERROR)) {
252         // accrueInterest emits logs on errors, but on top of that we want to log
253         // the fact that an attempted reduce reserves failed.
254         return fail(Error(error), FailureInfo.REDUCE_RESERVES_ACCRUE_INTEREST_FAILED
255             );
256     }
257
258     // If reserves were reduced in accrueInterest
259     if (reduceReservesBlockNumber == block.number) return (uint(Error.NO_ERROR));
260     // _reduceReservesFresh emits reserve-reduction-specific logs on errors, so we
261     // don't need to.
262     return _reduceReservesFresh(reduceAmount_);
263 }

```

Listing 3.3: VToken::_reduceReserves()

Recommendation Revisit the above revenue collection in VBNBAdmin so that it is consistent with other VToken markets.

Status This issue has been confirmed.

3.3 Incorrect Borrow Cap Enforcement in isolated-pools

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Comptroller
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The Venus protocol has a number of risk parameters that can be updated via governance proposals based on current market needs and community engagement. While examining a specific risk parameter `borrowCap`, we notice its enforcement needs to be revisited.

To elaborate, we show below the code snippet of the `Comptroller::preBorrowHook()` routine. This routine will kick in to ensure the new borrow will not result in the total borrow exceeding the `borrowCap`.

However, the current calculation of resulting total borrow does not take into consideration of possible bad debt (line 379). Therefore, the enforcement of `borrowCap` is less precise as desired.

```

349     function preBorrowHook(
350         address vToken,
351         address borrower,
352         uint256 borrowAmount
353     ) external override {
354         _checkActionPauseState(vToken, Action.BORROW);
355
356         if (!markets[vToken].isListed) {
357             revert MarketNotListed(address(vToken));
358         }
359
360         if (!markets[vToken].accountMembership[borrower]) {
361             // only vTokens may call borrowAllowed if borrower not in market
362             _checkSenderIs(vToken);
363
364             // attempt to add borrower to the market or revert
365             _addToMarket(VToken(msg.sender), borrower);
366         }
367
368         // Update the prices of tokens
369         updatePrices(borrower);
370
371         if (oracle.getUnderlyingPrice(vToken) == 0) {
372             revert PriceError(address(vToken));
373         }
374
375         uint256 borrowCap = borrowCaps[vToken];
376         // Skipping the cap check for uncapped coins to save some gas
377         if (borrowCap != type(uint256).max) {
378             uint256 totalBorrows = VToken(vToken).totalBorrows();
379             uint256 nextTotalBorrows = totalBorrows + borrowAmount;
380             if (nextTotalBorrows > borrowCap) {
381                 revert BorrowCapExceeded(vToken, borrowCap);
382             }
383         }
384         ...
385     }

```

Listing 3.4: `Comptroller::preBorrowHook()`

Recommendation Properly compute the resulting total borrow for the enforcement of the `borrowCap` risk parameter.

Status This issue has been fixed in the following commit: `f603afd`.

3.4 Removal of Unused Code

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: ProtocolShareReserve
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

The Venus protocol makes good use of a number of reference contracts, such as ERC20, SafeERC20, AccessControl, and Ownable, to facilitate its code implementation and organization. For example, the smart contract ProtocolShareReserve has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the ProtocolShareReserve contract, there is an unused event being defined, i.e., DestinationConfigured. This event may be safely removed. Moreover, the current ReserveHelpers contract defines a few functions that take an input argument IncomeType, which is not used yet.

```
67     /// @notice Event emitted after a income distribution target is configured
68     event DestinationConfigured(address indexed destination, uint percent, Schema schema
69         );
70
71     /// @notice Event emitted when asset is released to an target
72     event AssetReleased(
73         address indexed destination,
74         address indexed asset,
75         Schema schema,
76         uint256 percent,
77         uint256 amount
78     );
```

Listing 3.5: Certain Events Defined in ProtocolShareReserve

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been fixed in the following commit: [12d87b2](#).

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: RiskFund
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

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 this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (<https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729>).

```

194  /**
195   * @dev Approve the passed address to spend the specified amount of tokens on behalf
       of msg.sender.
196   * @param _spender The address which will spend the funds.
197   * @param _value The amount of tokens to be spent.
198   */
199   function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201       // To change the approve amount you first have to reduce the addresses '
202       // allowance to zero by calling 'approve(_spender, 0)' if it is not
203       // already 0 to mitigate the race condition described here:
204       // https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205       require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

207       allowed[msg.sender][_spender] = _value;
208       Approval(msg.sender, _spender, _value);
209   }

```

Listing 3.6: USDT Token Contract

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`. 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 `transfer()` as well, i.e., `safeTransfer()`.


```

38  /**
39   * @dev Deprecated. This function has issues similar to the ones found in
40   * {IERC20-approve}, and its usage is discouraged.
41   *
42   * Whenever possible, use {safeIncreaseAllowance} and
43   * {safeDecreaseAllowance} instead.
44   */
45  function safeApprove(
46      IERC20 token,
47      address spender,
48      uint256 value
49  ) internal {
50      // safeApprove should only be called when setting an initial allowance,
51      // or when resetting it to zero. To increase and decrease it, use
52      // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
53      require(
54          (value == 0) (token.allowance(address(this), spender) == 0),
55          "SafeERC20: approve from non-zero to non-zero allowance"
56      );
57      _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
58          spender, value));
59  }

```

Listing 3.7: SafeERC20::safeApprove()

In current implementation, if we examine the RiskFund::_swapAsset() routine that is designed to enable the token swap. To accommodate the specific idiosyncrasy, there is a need to use safeApprove(), instead of approve() (lines 292 – 292).

```

53  function _swapAsset(
54      VToken vToken,
55      address comptroller,
56      uint256 amountOutMin,
57      address[] calldata path
58  ) internal returns (uint256) {
59      require(amountOutMin != 0, "RiskFund: amountOutMin must be greater than 0 to
60          swap vToken");
61      require(amountOutMin >= minAmountToConvert, "RiskFund: amountOutMin should be
62          greater than minAmountToConvert");
63      uint256 totalAmount;
64
65      address underlyingAsset = vToken.underlying();
66      address convertibleBaseAsset_ = convertibleBaseAsset;
67      uint256 balanceOfUnderlyingAsset = poolsAssetsReserves[comptroller][
68          underlyingAsset];
69
70      ComptrollerViewInterface(comptroller).oracle().updatePrice(address(vToken));
71
72      uint256 underlyingAssetPrice = ComptrollerViewInterface(comptroller).oracle().
73          getUnderlyingPrice(
74              address(vToken)
75          );

```

```

72
73     if (balanceOfUnderlyingAsset > 0) {
74         Exp memory oraclePrice = Exp({ mantissa: underlyingAssetPrice });
75         uint256 amountInUsd = mul_ScalarTruncate(oraclePrice,
76             balanceOfUnderlyingAsset);
77
78         if (amountInUsd >= minAmountToConvert) {
79             assetsReserves[underlyingAsset] -= balanceOfUnderlyingAsset;
80             poolsAssetsReserves[comptroller][underlyingAsset] -=
81                 balanceOfUnderlyingAsset;
82
83             if (underlyingAsset != convertibleBaseAsset_) {
84                 require(path[0] == underlyingAsset, "RiskFund: swap path must start
85                     with the underlying asset");
86                 require(
87                     path[path.length - 1] == convertibleBaseAsset_,
88                     "RiskFund: finally path must be convertible base asset"
89                 );
90                 address pancakeSwapRouter_ = pancakeSwapRouter;
91                 IERC20Upgradeable(underlyingAsset).approve(pancakeSwapRouter_, 0);
92                 IERC20Upgradeable(underlyingAsset).approve(pancakeSwapRouter_,
93                     balanceOfUnderlyingAsset);
94                 uint256[] memory amounts = IPancakeswapV2Router(pancakeSwapRouter_).
95                     swapExactTokensForTokens(
96                         balanceOfUnderlyingAsset,
97                         amountOutMin,
98                         path,
99                         address(this),
100                         block.timestamp
101                     );
102                 totalAmount = amounts[path.length - 1];
103             } else {
104                 totalAmount = balanceOfUnderlyingAsset;
105             }
106         }
107     }
108     return totalAmount;
109 }

```

Listing 3.8: RiskFund::_swapAsset()

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

Status This issue has been fixed in the following commit: 6c559f1.

3.6 Inaccurate Transfer Event in isolated-pools/VToken

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: VToken
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

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

In the following, we use the `VToken` contract as an example. This contract has public functions that are used to transfer the `VToken`. While examining the `Transfer` events, we notice it is overloaded and the emitted information may be improved. Specifically, the `_seize()` routine is used to transfer certain amount of collateral tokens from the liquidated borrowed to the liquidator. We notice there are three `Transfer` events emitted at the end of the routine (lines 1225-1227). The first `Transfer` indicates the collateral transfer to the liquidator; the second shows the reduction of `totalSupply` as the collaterals are transferred out as the protocol income; and the last one intends to report the seized amount of the underlying token. Note that the second event is better to have `address(0)` as the recipient and the last one simply overloads the `Transfer` event, which is not appropriate.

```

1177     function _seize(
1178         address seizerContract,
1179         address liquidator,
1180         address borrower,
1181         uint256 seizeTokens
1182     ) internal {
1183         /* Fail if seize not allowed */
1184         comptroller.preSeizeHook(address(this), seizerContract, liquidator, borrower);

1186         /* Fail if borrower = liquidator */
1187         if (borrower == liquidator) {
1188             revert LiquidateSeizeLiquidatorIsBorrower();
1189         }

1191         /*
1192          * We calculate the new borrower and liquidator token balances, failing on
1193          * underflow/overflow:
1193          * borrowerTokensNew = accountTokens[borrower] - seizeTokens
1194          * liquidatorTokensNew = accountTokens[liquidator] + seizeTokens

```

```

1195     */
1196     uint256 liquidationIncentiveMantissa = ComptrollerViewInterface(address(
        comptroller))
1197     .liquidationIncentiveMantissa();
1198     uint256 numerator = mul_(seizeTokens, Exp({ mantissa: protocolSeizeShareMantissa
        }));
1199     uint256 protocolSeizeTokens = div_(numerator, Exp({ mantissa:
        liquidationIncentiveMantissa }));
1200     uint256 liquidatorSeizeTokens = seizeTokens - protocolSeizeTokens;
1201     Exp memory exchangeRate = Exp({ mantissa: _exchangeRateStored() });
1202     uint256 protocolSeizeAmount = mul_ScalarTruncate(exchangeRate,
        protocolSeizeTokens);

1204     //////////////////////////////////////
1205     // EFFECTS & INTERACTIONS
1206     // (No safe failures beyond this point)

1208     /* We write the calculated values into storage */
1209     totalSupply = totalSupply - protocolSeizeTokens;
1210     accountTokens[borrower] = accountTokens[borrower] - seizeTokens;
1211     accountTokens[liquidator] = accountTokens[liquidator] + liquidatorSeizeTokens;

1213     // _doTransferOut reverts if anything goes wrong, since we can't be sure if side
        effects occurred.
1214     // Transferring an underlying asset to the protocolShareReserve contract to
        channel the funds for different use.
1215     _doTransferOut(protocolShareReserve, protocolSeizeAmount);

1217     // Update the pool asset's state in the protocol share reserve for the above
        transfer.
1218     IProtocolShareReserve(protocolShareReserve).updateAssetsState(
1219         address(comptroller),
1220         underlying,
1221         IProtocolShareReserve.IncomeType.LIQUIDATION
1222     );

1224     /* Emit a Transfer event */
1225     emit Transfer(borrower, liquidator, liquidatorSeizeTokens);
1226     emit Transfer(borrower, address(this), protocolSeizeTokens);
1227     emit Transfer(address(this), protocolShareReserve, protocolSeizeAmount);
1228 }

```

Listing 3.9: VToken::_seize()

Recommendation Properly emit respective events when the liquidated collaterals are being seized.

Status This issue has been fixed in the following commits: 7ac357c and 4576435

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Automatic Income Allocation` support in `Venus`. This support allows for streamlined allocation and distribution of protocol income, either spread of liquidation. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-563: Assignment to Variable without Use. <https://cwe.mitre.org/data/definitions/563.html>.
- [3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [4] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [5] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [6] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [7] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [8] PeckShield. PeckShield Inc. <https://www.peckshield.com>.