SMART CONTRACT AUDIT REPORT

for

Venus Liquidator

Prepared By: Xiaomi Huang

PeckShield

July 5, 2023

Document Properties

Client	Venus
Title	Smart Contract Audit Report
Target	Venus Liquidator
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 5, 2023	Xuxian Jiang	Final Release
1.0-rc	June 18, 2023	Xuxian Jiang	Release Candidate

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 new Venus Liquidator contract, 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About Venus Liquidator

The Venus protocol is designed to enable a complete algorithmic money market protocol on Binance Smart Chain (BSC). Venus enables users to utilize their cryptocurrencies by supplying collateral to the protocol that may be borrowed by pledging over-collateralized cryptocurrencies. It also features a synthetic stablecoin (VAI) that is not backed by a basket of fiat currencies but by a basket of cryptocurrencies. Venus utilizes the BSC for fast, low-cost transactions while accessing a deep network of wrapped tokens and liquidity. The new Venus Liquidator integrates the AccessControlManager support, redeems the liquidated collateral to ProtocolShareReserve, as well as forces the liquidation of VAI positions first before other positions. The basic information of the Venus Liquidator feature is as follows:

Table 1.1: Basic Information of Venus Liquidator

Item	Description
Name	Venus
Website	https://venus.io/
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 5, 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/venus-protocol/pull/241 (9ee9738)

1.2 About PeckShield

PeckShield Inc. [7] 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 the OWASP Risk Rating Methodology [6]:

- 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 Audit Checklist

Category	Checklist Items
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 checklist of 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) [5], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	2	■ ■
Informational	1	■
Total	3	

We have previously audited the main Venus protocol. In this report, we exclusively focus on the specific pull request `PR-241`, we determine three issues 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 discussion of the issues 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, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Venus Liquidator Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Revisited Splitting of Liquidation Incentive	Business Logic	Resolved
PVE-002	Low	Improved Consistency/Gas in Coding Style	Coding Practices	Resolved
PVE-003	Low	Trust on Admin Keys	Security Features	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Revisited Splitting of Liquidation Incentive

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Liquidator`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The new `Liquidator` contract shares the liquidation incentive between the liquidator and the protocol-specified `ProtocolShareReserve` account. The share between the two is determined by a protocol parameter `treasuryPercentMantissa`. While examining the use of this specific `treasuryPercentMantissa` parameter, we notice the need to revisit the incentive-splitting logic.

In particular, we show below two related routines: `_splitLiquidationIncentive()` and `validateTreasuryPercentMantissa()`. The first routine executes the incentive splitting based on the `treasuryPercentMantissa / comptroller.liquidationIncentiveMantissa()` percentage. However, from the second routine, we notice `treasuryPercentMantissa` is validated to be no larger than `comptroller.liquidationIncentiveMantissa()- 1e18`. As a result, the current incentive-splitting logic can be improved by using the `treasuryPercentMantissa / (comptroller.liquidationIncentiveMantissa()- 1e18)` percentage.

```
    /// @dev Computes the amounts that would go to treasury and to the liquidator.
    function _splitLiquidationIncentive(uint256 seizedAmount) internal view returns (
        uint256 ours, uint256 theirs) {
        uint256 totalIncentive = comptroller.liquidationIncentiveMantissa();
        ours = (seizedAmount * treasuryPercentMantissa) / totalIncentive;
        theirs = seizedAmount - ours;
        return (ours, theirs);
    }
```

Listing 3.1: `Liquidator::_splitLiquidationIncentive()`

```
    function validateTreasuryPercentMantissa(uint256 treasuryPercentMantissa_) internal view {
        uint256 maxTreasuryPercentMantissa = comptroller.liquidationIncentiveMantissa() - 1e18;
        if (treasuryPercentMantissa_ > maxTreasuryPercentMantissa) {
            revert TreasuryPercentTooHigh(maxTreasuryPercentMantissa, treasuryPercentMantissa_);
        }
    }
```

Listing 3.2: `Liquidator::validateTreasuryPercentMantissa()`

Recommendation Revisit the incentive-splitting logic by using the intended `treasuryPercentMantissa`
.

Status The issue has been resolved as the team confirms it is part of the design.

3.2 Improved Consistency/Gas in Coding Style

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Liquidator`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `Venus` protocol was originally forked from `CompoundV2` with innovative customization. One preferred coding style in `CompoundV2` and `Venus` is the use of proper naming conventions, including a set of rules for choosing names for variables, functions, and other important entities. One specific one is the prefixed/postfixed use of `underscore` in local variables and function arguments. While examining the current coding style, we notice certain inconsistencies that can be better resolved.

As an example, we show below two functions: `setMinLiquidatableVAI()` and `setPendingRedeemChunkLength()`. As their names indicate, the former is used to set the minimum amount threshold in `VAI` liquidation and the latter is used to configure the `pendingRedeem` array length used in liquidation redemption. However, they exhibit different coding styles in naming the function arguments: the former uses the `underscore` prefix while the later instead uses the `underscore` postfix. Also, the function `reduceReservesInternal()` chooses the `underscore` postfix for the local variable `pendingReedemLength_`. For consistency, we suggest the `underscore` postfix in the function arguments while the `underscore` prefix for local variables.

```
    /**
     * @notice Sets the threshold for minimum amount of vaiLiquidate
```

```
552      * @param _minLiquidatableVAI New address for the access control
553      */
554     function setMinLiquidatableVAI(uint256 _minLiquidatableVAI) external {
555         _checkAccessAllowed("setMinLiquidatableVAI(uint256)");
556         uint256 oldMinLiquidatableVAI_ = minLiquidatableVAI;
557         minLiquidatableVAI = _minLiquidatableVAI;
558         emit NewMinLiquidatableVAI(oldMinLiquidatableVAI_, _minLiquidatableVAI);
559     }
560
561     /**
562      * @notice Length of the pendingRedeem array to be consider while redeeming in
           Liquidation transaction
563      * @param newLength_ Length of the chunk
564      */
565     function setPendingRedeemChunkLength(uint256 newLength_) external {
566         _checkAccessAllowed("setPendingRedeemChunkLength(uint256)");
567         uint256 oldPendingRedeemChunkLength_ = pendingRedeemChunkLength;
568         pendingRedeemChunkLength = newLength_;
569         emit NewPendingRedeemChunkLength(oldPendingRedeemChunkLength_,
                pendingRedeemChunkLength);
570     }
```

Listing 3.3: `Liquidator::setMinLiquidatableVAI()/setPendingRedeemChunkLength()`

Moreover, in the above `setPendingRedeemChunkLength()`, the `NewPendingRedeemChunkLength` event can be emitted with `emit NewPendingRedeemChunkLength(oldPendingRedeemChunkLength_, newLength_)` (line 569) to save one storage read.

Recommendation Resolve the above-mentioned inconsistency in the coding style.

Status This issue has been fixed in the following commit: 92ff328.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Liquidator`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the new `Liquidator` contract, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and access control adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    /**
     * @notice Sets the address of the access control of this contract
     * @dev Admin function to set the access control address
     * @param newAccessControlAddress New address for the access control
     */
    function setAccessControl(address newAccessControlAddress) external onlyOwner {
        _setAccessControlManager(newAccessControlAddress);
    }

    /**
     * @notice Sets protocol share reserve contract address
     * @param protocolShareReserve_ The address of the protocol share reserve contract
     */
    function setProtocolShareReserve(address payable protocolShareReserve_) external
        onlyOwner {
        _setProtocolShareReserve(protocolShareReserve_);
    }
```

Listing 3.4: Example Privileged Operations in the `Liquidator` Contract

If the privileged `owner` account is managed by a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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

Status This issue has been resolved as the `owner` is managed by the `Venus: Timelock` contract deployed at `0x939bd8d64c0a9583a7dcea9933f7b21697ab6396`.

4 Conclusion

In this audit, we have analyzed the new `Venus Liquidator` design and implementation. The system presents a unique, robust offering as a decentralized money market protocol with both secure lending and synthetic stablecoins. The new `Venus Liquidator` integrates the `AccessControlManager` support, redeems the liquidated collateral to `ProtocolShareReserve`, as well as forces the liquidation of `VAI` positions first before other positions. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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