SMART CONTRACT AUDIT REPORT

for

Venus BUSDLiquidator

Prepared By: Xiaomi Huang

PeckShield

October 20, 2023

Document Properties

Client	Venus
Title	Smart Contract Audit Report
Target	Venus BUSDLiquidator
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 20, 2023	Xuxian Jiang	Final Release
1.0-rc	October 7, 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 BUSDLiquidator support in Venus, 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 BUSDLiquidator

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. The audited Venus BUSDLiquidator support adds a custom mechanism for BUSD liquidations, i.e., only BUSDLiquidator will be allowed to liquidate BUSD borrows. To reduce the potential impact, BUSDLiquidator will be allowed to pause and unpause liquidating BUSD debts. Between the transactions, BUSD liquidations will be paused, as far as Comptroller is concerned. The basic information of the Venus BUSDLiquidator feature is as follows:

Table 1.1: Basic Information of Venus BUSDLiquidator

Item	Description
Name	Venus
Website	https://venus.io/
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 20, 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/362 (90aa99d)

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 BUSDLiquidator` 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	1	■
Informational	1	■
Total	2	

We have previously audited the main Venus protocol. In this report, we exclusively focus on the specific pull request `PR-362`, we determine two 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 low-severity vulnerability and 1 informational recommendation.

Table 2.1: Key Venus BUSDLiquidator Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Redundant Token Approval Removal in BUSDLiquidator	Coding Practices	Resolved
PVE-002	Low	Revisited vTokenCollateral Validation in Liquidator	Business Logic	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 Redundant Token Approval Removal in BUSDLiquidator

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BUSDLiquidator`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

By design, the new `BUSDLiquidator` contract will be allowed to liquidate `BUSD` borrows. While examining the actual `BUSD` liquidation logic, we notice current implementation can be improved to reduce an unnecessary token approval.

In particular, we show below the implementation of the affected routine – `_liquidateBorrow()`. This routine performs the actual liquidation by transferring `BUSD` from the sender to this contract, repaying the debt, and transferring the seized collateral to the sender and the treasury. While repaying the debt, there is a need to approve the repayment to `liquidatorContract`. We notice current implementation has properly revoked the token approval at end of the liquidation (line 132), which means the initial approval reset (line 127) becomes redundant. As an alternative, we can also only keep the initial approval reset (line 127) and remove the second token approval reset (line 132).

```
122     function _liquidateBorrow(address borrower, uint256 repayAmount, IVToken
            vTokenCollateral) internal {
123         ILiquidator liquidatorContract = ILiquidator(comptroller.liquidatorContract());
124         IERC20Upgradeable busd = IERC20Upgradeable(vBUSD.underlying());
125
126         uint256 actualRepayAmount = _transferIn(busd, msg.sender, repayAmount);
127         approveOrRevert(busd, address(liquidatorContract), 0);
128         approveOrRevert(busd, address(liquidatorContract), actualRepayAmount);
129         uint256 balanceBefore = vTokenCollateral.balanceOf(address(this));
130         liquidatorContract.liquidateBorrow(address(vBUSD), borrower, actualRepayAmount,
                vTokenCollateral);
```

```
        uint256 receivedAmount = vTokenCollateral.balanceOf(address(this)) -
            balanceBefore;
        approveOrRevert(busd, address(liquidatorContract), 0);

        (uint256 liquidatorAmount, uint256 treasuryAmount) = _computeShares(
            receivedAmount);
        vTokenCollateral.safeTransfer(msg.sender, liquidatorAmount);
        vTokenCollateral.safeTransfer(treasury, treasuryAmount);
    }
```

Listing 3.1: `BUSDLiquidator::_liquidateBorrow()`

Recommendation Revisit the above routine to only keep one token approval reset operation.

Status This issue has been fixed in the following commit: 592b022.

3.2 Revisited vTokenCollateral Validation in Liquidator

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

Description

To facilitate the liquidation process, the `Venus` protocol has a dedicated `Liquidator` contract. While reviewing its interaction with various `Venus` markets, we notice the liquidation logic can be improved by applying additional validation on the user input.

In the following, we show the implementation of actual liquidation routine `liquidateBorrow()`. This routine has four arguments: `vToken`, `borrower`, `repayAmount`, and `vTokenCollateral`. While these four arguments are self-evident, we notice the last argument can be better validated to ensure that `borrower` has indeed entered the `vTokenCollateral` market via `require(comptroller.markets(vTokenCollateral).accountMembership(borrower))`. The reason is that current liquidation code allows to seize the collateral from the market the borrower has not entered yet.

```
    function liquidateBorrow(
        address vToken,
        address borrower,
        uint256 repayAmount,
        IVToken vTokenCollateral
    ) external payable nonReentrant {
        ensureNonzeroAddress(borrower);
        checkRestrictions(borrower, msg.sender);
        uint256 ourBalanceBefore = vTokenCollateral.balanceOf(address(this));
```

```
        if (vToken == address(vBnb)) {
            if (repayAmount != msg.value) {
                revert WrongTransactionAmount(repayAmount, msg.value);
            }
            vBnb.liquidateBorrow{ value: msg.value }(borrower, vTokenCollateral);
        } else {
            if (msg.value != 0) {
                revert WrongTransactionAmount(0, msg.value);
            }
            if (vToken == address(vaiController)) {
                _liquidateVAI(borrower, repayAmount, vTokenCollateral);
            } else {
                _liquidateBep20(IVBep20(vToken), borrower, repayAmount, vTokenCollateral
                    );
            }
        }
        uint256 ourBalanceAfter = vTokenCollateral.balanceOf(address(this));
        uint256 seizedAmount = ourBalanceAfter - ourBalanceBefore;
        (uint256 ours, uint256 theirs) = _distributeLiquidationIncentive(
            vTokenCollateral, seizedAmount);
        emit LiquidateBorrowedTokens(
            msg.sender,
            borrower,
            repayAmount,
            vToken,
            address(vTokenCollateral),
            ours,
            theirs
        );
    }
```

Listing 3.2: `Liquidator::liquidateBorrow()`

Recommendation Revisit the above routine to ensure the liquidation is only allowed to seize the collateral from the market the borrower has entered before.

Status This issue has been fixed in the following commit: 05ff797.

4 Conclusion

In this audit, we have analyzed the new `Venus BUSDLiquidator` design and implementation. The system presents a unique, robust offering as a decentralized money market protocol with both secure lending and synthetic stablecoins. The audited `Venus BUSDLiquidator` support adds a custom mechanism for `BUSD` liquidations, i.e., only `BUSDLiquidator` will be allowed to liquidate `BUSD` borrows. To reduce the potential impact, `BUSDLiquidator` will be allowed to pause and unpause liquidating `BUSD` debts. Between the transactions, `BUSD` liquidations will be paused, as far as `Comptroller` is concerned. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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