SMART CONTRACT AUDIT REPORT

for

SwapRouter in Venus

Prepared By: Xiaomi Huang

PeckShield

April 19, 2023

Document Properties

Client	Venus
Title	Smart Contract Audit Report
Target	SwapRouter
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	April 19, 2023	Xiaotao Wu	Final Release
1.0-rc	April 6, 2023	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 Venus SwapRouter feature, 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 SwapRouter

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 SwapRouter feature allows anyone to repay their Venus debt or supply funds to a Venus market, using a different coin of the involved market. Moreover, the SwapRouter contract adds similar functions as PancakeSwap, with the goal of identifying swaps originating from the Venus protocol. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The SwapRouter

Item	Description
Name	Venus
Website	https://venus.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 19, 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/tree/develop/contracts/Swap (754240d)

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

- https://github.com/VenusProtocol/venus-protocol/tree/develop/contracts/Swap (a438094)

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 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 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) [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.

1.4 Disclaimer

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

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

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

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Venus SwapRouter` feature. 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	1	■
Low	0	
Informational	1	■
Total	2	

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 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 and 1 informational recommendation.

Table 2.1: Key SwapRouter Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Slippage Control for Supporting Fee Tokens	Time and State	Fixed
PVE-002	Informational	Improved Sanity Checks For System Parameters	Coding Practices	Fixed

Besides 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 Slippage Control for Supporting Fee Tokens

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SwapRouter`
- Category: Time and State [4]
- CWE subcategory: CWE-682 [2]

Description

The `SwapRouter` contract is designed to facilitate the user operations with `Venus` in repaying debt or supplying funds to a `Venus` market, using a different coin of the involved market. While examining the current swap logic, we notice the slippage control for deflationary token is not correct and need to be revisited.

In the following, we use the `swapAndSupplyAtSupportingFee()` routine as an example and show below the related code snippet. In the `swapAndSupplyAtSupportingFee()` routine, the amount of tokens to be swapped is specified by the input `amountIn` parameter and the amount of tokens to be received is calculated according to `amountIn` (`amounts = PancakeLibrary.getAmountsOut(factory, amountIn, path)`). To prevent possible front-running attacks, we notice it restricts the amount of received tokens should not be less than `amountOutMin`. However, it ignores the fact that deflationary tokens may charge a certain fee for every `transfer()` or `transferFrom()` operation. In other words, the amount of received tokens will be less than `amounts[amounts.length - 1]`, which opens up the possibility for front-running attacks, resulting in a smaller gain for this round of swap. Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell.

```
    function swapAndSupplyAtSupportingFee(
        address vTokenAddress,
```

```
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        uint256 deadline
    ) external override ensure(deadline) ensureVTokenListed(vTokenAddress) {
        uint256 balanceBefore = IERC20(path[path.length - 1]).balanceOf(address(this));
        _swapExactTokensForTokens(amountIn, amountOutMin, path, address(this),
            TypesOfTokens.SUPPORTING_FEE);
        uint256 balanceAfter = IERC20(path[path.length - 1]).balanceOf(address(this));
        uint256 swapAmount = balanceAfter - balanceBefore;
        _supply(path[path.length - 1], vTokenAddress, swapAmount);
    }
```

Listing 3.1: SwapRouter::swapAndSupplyAtSupportingFee()

```
    function _swapExactTokensForTokens(
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        TypesOfTokens swapFor
    ) internal returns (uint256[] memory amounts) {
        amounts = PancakeLibrary.getAmountsOut(factory, amountIn, path);
        if (amounts[amounts.length - 1] < amountOutMin) {
            revert OutputAmountBelowMinimum(amounts[amounts.length - 1], amountOutMin);
        }
        address pairAddress = PancakeLibrary.pairFor(factory, path[0], path[1]);
        TransferHelper.safeTransferFrom(path[0], msg.sender, pairAddress, amounts[0]);
        if (swapFor == TypesOfTokens.NON_SUPPORTING_FEE) {
            _swap(amounts, path, to);
        } else {
            _swapSupportingFeeOnTransferTokens(path, to);
        }
        emit SwapTokensForTokens(msg.sender, path, amounts);
    }
```

Listing 3.2: RouterHelper::_swapExactTokensForTokens()

Note the similar issue also exists in all the supporting fee related routines of the same contract.

Recommendation Revisit the above-mentioned routines by adding effective slippage control.

Status The issue has been fixed by the following pull request: 243.

3.2 Improved Sanity Checks For System Parameters

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: SwapRouter
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned in Section 3.1, the SwapRouter contract allows anyone to repay their Venus debt or supply funds to a Venus market, using a different coin of the involved market. While reviewing the current SwapRouter contract, we notice some of existing functions can benefit from additional sanity checks.

To elaborate, we use the swapAndSupply() routine as an example and show the related code snippet below. Specifically, there is a lack of length verification for the input path parameter. If the path.length is equal to 0, the execution in line 88 will revert.

```
81     function swapAndSupply(
82         address vTokenAddress,
83         uint256 amountIn,
84         uint256 amountOutMin,
85         address[] calldata path,
86         uint256 deadline
87     ) external override ensure(deadline) ensureVTokenListed(vTokenAddress) {
88         uint256 balanceBefore = IERC20(path[path.length - 1]).balanceOf(address(this));
89         _swapExactTokensForTokens(amountIn, amountOutMin, path, address(this),
               TypesOfTokens.NON_SUPPORTING_FEE);
90         uint256 balanceAfter = IERC20(path[path.length - 1]).balanceOf(address(this));
91         uint256 swapAmount = balanceAfter - balanceBefore;
92         _supply(path[path.length - 1], vTokenAddress, swapAmount);
93     }
```

Listing 3.3: SwapRouter::swapAndSupply()

Note the similar issue also exists in all the supply and repay related routines of the same contract.

Recommendation Add necessary sanity checks to ensure the length of the input path parameter is greater than 1 for the above mentioned functions.

Status The issue has been fixed by the following pull request: 243.

4 Conclusion

In this audit, we have analyzed the `Venus SwapRouter` design and implementation. The audited `Venus SwapRouter` feature allows anyone to repay their `Venus` debt or supply funds to a `Venus` market, using a different coin of the involved market. Moreover, the `SwapRouter` contract adds similar functions as `PancakeSwap`, with the goal of identifying swaps originating from the `Venus` protocol. 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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

[4] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.