SMART CONTRACT AUDIT REPORT

for

Venus Grant

Prepared By: Yiqun Chen

PeckShield

June 12, 2021

Document Properties

Client	Venus
Title	Smart Contract Audit Report
Target	Venus Grant
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 12, 2021	Xiaotao Wu	Final Release
1.0-rc	June 7, 2021	Xiaotao Wu	Release Candidate #1
0.1	June 1, 2021	Xiaotao Wu	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

PeckShield

1 Introduction

Given the opportunity to review the **Venus Grant** design document and related smart contract source code, 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 an issue related to either security or performance. This document outlines our audit results.

1.1 About Venus Grant

The `Venus` protocol is designed to enable a complete algorithmic money market protocol on `Binance Smart Chain (BSC)`. The protocol designs are architected and forked based on `Compound` and `MakerDAO` and synced into the `Venus` platform to capitalize the benefits of both systems. `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 audited `Venus Grant` support allows for customizing rewards for protocol contributors.

The basic information of Venus Grant is as follows:

Table 1.1: Basic Information of Venus Grant

Item	Description
Issuer	Venus
Website	https://venus.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 12, 2021

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.git (5b7f6af)

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

- https://github.com/VenusProtocol/venus-protocol.git (ec22556)

1.2 About PeckShield

PeckShield Inc. [5] 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 [4]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

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

To evaluate the risk, we go through a 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) [3], 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

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

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.

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the Venus Grant support. 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	0	
Total	1	

We have previously audited the main Venus protocol. In this report, we exclusively focus on the specific pull request `5b7f6af`, we determine one issue of low severity that needs to be brought up and paid more attention to, which is categorized in the above table. More information can be found in the next subsection, and the detailed discussion of this issue is 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 issue (shown in Table 2.1), including 1 low-severity vulnerability.

Table 2.1: Key Venus Grant Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Logic Issue of Comptroller::_setContributorVenusSpeed()	Business Logics	Fixed

Beside the identified issue, 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.

PeckShield

3 | Detailed Results

3.1 Logic Issue of Comptroller::_setContributorVenusSpeed()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Comptroller
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

The Venus Grant support requires the ability of customizing the distribution speed of the governance token to certain contributors. And the support is mainly implemented in the Comptroller contract with the _setContributorVenusSpeed() function. While examining this function, we notice a minor issue that may need to be addressed.

To elaborate, we show below the _setContributorVenusSpeed() function. When there is a need to halt liquidity rewards for the given contributor, this function can be invoked by specifying 0 as its venusSpeed. Current implementation intends to release the contributor's storage slot (line 1419) when the corresponding liquidity reward is halted. Note that the storage slot is used to store the last block information at which a contributor's XVS rewards have been allocated.

```
1407     /**
1408      * @notice Set Venus speed for a single contributor
1409      * @param contributor The contributor whose Venus speed to update
1410      * @param venusSpeed New Venus speed for contributor
1411      */
1412     function _setContributorVenusSpeed(address contributor, uint venusSpeed) public {
1413         require(adminOrInitializing(), "only admin can set xvs speed");
1414 
1415         // note that Venus speed could be set to 0 to halt liquidity rewards for a
             contributor
1416         updateContributorRewards(contributor);
1417         if (venusSpeed == 0) {
1418             // release storage
```

```
1419             delete lastContributorBlock[contributor];
1420         }
1421         lastContributorBlock[contributor] = getBlockNumber();
1422         venusContributorSpeeds[contributor] = venusSpeed;
1423 
1424         emit ContributorVenusSpeedUpdated(contributor, venusSpeed);
1425     }
```

Listing 3.1: `Comptroller::_setContributorVenusSpeed()`

However, after the storage slot is released, the current implementation reuses this storage slot (line 1421) to store the current block information for the contributor without judging whether this contributor's `venusSpeed` is not equal to 0.

Recommendation Update the contributor's block information only when this contributor's `venusSpeed` is not equal to 0.

Status The issue has been addressed by the following commit: `ec22556`.

4 Conclusion

In this audit, we have analyzed the `Venus Grant` design and implementation. The system presents a unique, robust offering as a decentralized money market protocol with both secure lending and synthetic stablecoins. The protocol designs are architected and forked based on `Compound` and `MakerDAO` and synced into the `Venus` platform to capitalize the benefits of both systems. The audited `Venus Grant` support allows for customizing rewards for protocol contributors. 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.

PeckShield

References

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

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

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

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

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

PeckShield