SMART CONTRACT AUDIT REPORT

for

Venus Prime

Prepared By: Xiaomi Huang

PeckShield

August 26, 2023

Document Properties

Client	Venus
Title	Smart Contract Audit Report
Target	Venus Prime
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 26, 2023	Xuxian Jiang	Final Release
1.0-rc	August 22, 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 Prime . . . 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 Incorrect Prime Token Burn Logic in _burn() . . . 11

3.2 Removal of Unused Code . . . 12

3.3 Timely executeBoost() Before Prime Score Update . . . 14

3.4 Trust Issue of Admin Keys . . . 15

4 Conclusion **17**

References **18**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Venus Prime` 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 Prime

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 `Venus Prime` support allows for the income distribution from boosted markets to the prime token holders in real-time. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Venus Prime

Item	Description
Name	Venus
Website	https://venus.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 26, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. This audit covers on the changes from `PR #196`: `https://github.com/VenusProtocol/venus-protocol/pull/196/`.

- https://github.com/VenusProtocol/venus-protocol.git (44a5411)

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

- https://github.com/VenusProtocol/venus-protocol.git (f31a054)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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) [7], 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 Prime` protocol. 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	2	■■
Low	2	■■
Informational	0	
Total	4	

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 2 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Venus Prime Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Prime Token Burn Logic in _burn()	Business Logic	Resolved
PVE-002	Low	Redundant Prime updateScore in PolicyFacet::repayBorrowAllowed()	Coding Practices	Resolved
PVE-003	Medium	Timely executeBoost() Before Prime Score Update	Business Logic	Resolved
PVE-004	Low	Trust Issue of Admin Keys	Security Features	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 Incorrect Prime Token Burn Logic in _burn()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Prime
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

There are two types of Venus Prime tokens: revocable and irrevocable. While both may be issued, users can only mint revocable prime tokens. And there are limits to the total number of revocable and irrevocable prime tokens that can be minted. While examining the current token-burning logic, we notice the logic needs improvement.

In the following, we show the code snippet from the related _burn() routine. When a prime token is burned, the related exists field is updated to be false. However, the total number of _totalIrrevocable or _totalRevocable is updated based on the tokens[user].isIrrevocable field, which has been prematurely updated to be false. In other words, its update needs to be performed after updating the total number of _totalIrrevocable or _totalRevocable.

```
    function _burn(address user) internal {
        if (!tokens[user].exists) revert UserHasNoPrimeToken();

        tokens[user].exists = false;
        tokens[user].isIrrevocable = false;

        if (tokens[user].isIrrevocable) {
            _totalIrrevocable--;
        } else {
            _totalRevocable--;
        }

        _updateRoundAfterTokenBurned();
```

```

        emit Burn(user);
    }
```

Listing 3.1: `Prime::_burn()`

Moreover, the subroutine of `_updateRoundAfterTokenBurned()` also needs revision. Specifically, if the given `user` already updated in `isScoreUpdated[nextScoreUpdateRoundId][user]`, there is no need to perform `pendingScoreUpdates--` (line 562), even though there is always a need to perform `totalScoreUpdatesRequired--` (line 561).

```
556     /**
557      * @notice update the required score updates when token is burned before round is
            completed
558      */
559     function _updateRoundAfterTokenBurned() internal {
560         if (pendingScoreUpdates > 0) {
561             totalScoreUpdatesRequired--;
562             pendingScoreUpdates--;
563         }
564     }
```

Listing 3.2: `Prime::_updateRoundAfterTokenBurned()`

Recommendation Revisit the above `_burn()` logic to properly burn a prime token.

Status This issue has been fixed in the following commits: `05bcd40` and `3128c23`.

3.2 Removal of Unused Code

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Prime`, `PolicyFacet`
- Category: Coding Practices [5]
- 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 `Prime` 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 `Prime` contract, there is an unused interface being defined, i.e., `ERC20Interface`. This interface may be safely removed. Moreover, the current `PrimeStorageV1` contract defines a constant `INCOME_DISTRIBUTION_BPS`, which is not used yet.

```
interface IVToken {
    function borrowBalanceStored(address account) external returns (uint);
    function exchangeRateStored() external returns (uint);
    function balanceOf(address account) external view returns (uint);
    function underlying() external view returns (address);
}

interface ERC20Interface {
    function decimals() external view returns (uint8);
}
```

Listing 3.3: Certain Interfaces Defined in `Prime`

In addition, we notice the function `PolicyFacet::repayBorrowAllowed()` is revised to invoke `prime.updateScore(borrower, vToken)` (line 193), which is not necessary as the same routine will be invoked inside `PolicyFacet::repayBorrowVerify()`. The same is also potentially applicable to `PolicyFacet::seizeVerify()`.

```
175     function repayBorrowAllowed(
176         address vToken,
177         // solhint-disable-next-line no-unused-vars
178         address payer,
179         address borrower,
180         // solhint-disable-next-line no-unused-vars
181         uint256 repayAmount
182     ) external returns (uint256) {
183         checkProtocolPauseState();
184         checkActionPauseState(vToken, Action.REPAY);
185         ensureListed(markets[vToken]);
186 
187         // Keep the flywheel moving
188         Exp memory borrowIndex = Exp({ mantissa: VToken(vToken).borrowIndex() });
189         updateVenusBorrowIndex(vToken, borrowIndex);
190         distributeBorrowerVenus(vToken, borrower, borrowIndex);
191 
192         if (address(prime) != address(0)) {
193             prime.updateScore(borrower, vToken);
194         }
195 
196         return uint256(Error.NO_ERROR);
197     }
```

Listing 3.4: `PolicyFacet::repayBorrowAllowed()`

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

Status This issue has been fixed in the following commits: `c9e0809` and `34c3ea5`.

3.3 Timely executeBoost() Before Prime Score Update

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PolicyFacet`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Venus Prime` protocol needs to timely keep track of user score and rewards according to the `Goldfinch rewards` mechanism. For this purpose, the protocol has defined global states, i.e., `rewardIndex` and `sumOfMembersScore`, for each supported market. The former needs to be updated whenever a user's staked `XVS` or `supply/borrow` changes while the latter represents the current sum of all the prime token holders score.

To elaborate, we show below the code snippet of the `PolicyFacet::repayBorrowVerify()` routine. This routine will kick in to ensure the `sumOfMembersScore` state is timely updated for each borrow/-supply change. However, there is also a need to timely call `executeBoost()` to update `rewardIndex` for the user and the market. Note this issue affects a number of routines, including `mintVerify()`, `redeemVerify()`, `borrowVerify()`, `repayBorrowVerify()`, `liquidateBorrowVerify()`, `seizeVerify()`, and `transferVerify()`.

```
    function repayBorrowVerify(
        address vToken,
        address payer,
        address borrower,
        uint256 actualRepayAmount,
        uint256 borrowerIndex
    ) external {
        if (address(prime) != address(0)) {
            prime.updateScore(borrower, vToken);
        }
    }
```

Listing 3.5: `PolicyFacet::repayBorrowVerify()`

Recommendation Timely update `rewardIndex` for the user and the market before updating the market's `sumOfMembersScore`.

Status This issue has been fixed in the following commits: `03839e5` and `f31a054`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Prime`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the new `Prime` contract, there is a privileged access control module, i.e., `AccessControlledV8`, which plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and role authorization). It also has the privilege to affect 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.

```
    function updateMultipliers(address market, uint256 _supplyMultiplier, uint256
        _borrowMultiplier) external {
        _checkAccessAllowed("updateMultipliers(address,uint256,uint256)");
        if (!markets[market].exists) revert MarketNotSupported();

        accrueInterest(market);
        markets[market].supplyMultiplier = _supplyMultiplier;
        markets[market].borrowMultiplier = _borrowMultiplier;

        _startScoreUpdateRound();
    }

    /**
     * @notice Add a market to prime program
     * @param vToken address of the market vToken
     * @param supplyMultiplier the multiplier for supply cap. It should be converted to
       1e18
     * @param borrowMultiplier the multiplier for borrow cap. It should be converted to
       1e18
     */
    function addMarket(address vToken, uint256 supplyMultiplier, uint256
        borrowMultiplier) external {
        _checkAccessAllowed("addMarket(address,uint256,uint256)");
        if (markets[vToken].exists) revert MarketAlreadyExists();

        markets[vToken].rewardIndex = 0;
        markets[vToken].supplyMultiplier = supplyMultiplier;
        markets[vToken].borrowMultiplier = borrowMultiplier;
        markets[vToken].sumOfMembersScore = 0;
        markets[vToken].exists = true;

        vTokenForAsset[_getUnderlying(vToken)] = vToken;
```

```

        allMarkets.push(vToken);
        _startScoreUpdateRound();
    }
```

Listing 3.6: Example Privileged Operations in the `Liquidator` Contract

If the privileged admins are 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 design and implementation of the `Venus Prime` protocol, which allows for the income distribution from boosted markets to the prime token holders in real-time. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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

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