SMART CONTRACT AUDIT REPORT

for

Venus (Multichain)

Prepared By: Xiaomi Huang

PeckShield

October 20, 2023

Document Properties

Client	Venus
Title	Smart Contract Audit Report
Target	Venus Multichain
Version	1.0
Author	Xuxian Jiang
Auditors	Jianzuo Shen, 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 19, 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

Given the opportunity to review the design document and related smart contract source code of the multichain 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 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

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 multichain support aims to expand into other blockchains, including `Ethereum mainnet`, `Arbitrum`, `Polygon zvEVM`, and `opBNB`, while still maintaining seamless user experience. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Venus Multichain

Item	Description
Name	Venus
Website	https://venus.io/
Type	EVM 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 `PRs` used in this audit. This audit covers the following changes:

- https://github.com/VenusProtocol/venus-protocol/pull/345 (`PR #345`)

- https://github.com/VenusProtocol/isolated-pools/pull/291 (PR #291)
- https://github.com/VenusProtocol/isolated-pools/pull/294 (PR #294)
- https://github.com/VenusProtocol/oracle/pull/124 (PR #124)

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 multichain support in `Venus`. 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	0	
Low	2	■■
Informational	2	■■
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 low-severity vulnerabilities and 2 informational suggestions.

Table 2.1: Key Venus Multichain Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Improved Gas Efficiency in TokenController	Coding Practices	Resolved
PVE-002	Informational	Inconsistent NatSpec Comments in BaseXVSProxyOFT	Coding Practices	Resolved
PVE-003	Low	Removal of payable Modifier in XVSBridgeAdmin	Coding Practices	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 Improved Gas Efficiency in TokenController

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `TokenController`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

The audited multichain support has a built-in `TokenController` contract that acts as the governance and access control mechanism. While examining current admin-related operations, we notice an internal helper can be improved.

In the following, we show the code snippet from the related helper: `_isEligibleToMint()`. This helper is designed to check the minter cap of the given `from_` as well as the eligibility of the intended receiver (`to_`) to receive tokens. We notice this helper will emit an event `MintLimitDecreased` with the `availableLimit` information, which can be computed as `mintingCap - totalMintedNew`, saving one `SLOAD` when compared with the existing approach (line 158).

```
146     function _isEligibleToMint(address from_, address to_, uint256 amount_) internal {
147         if (_blacklist[to_]) {
148             revert MintNotAllowed(from_, to_);
149         }
150         uint256 mintingCap = minterToCap[from_];
151         uint256 totalMintedOld = minterToMintedAmount[from_];
152         uint256 totalMintedNew = totalMintedOld + amount_;
153 
154         if (totalMintedNew > mintingCap) {
155             revert MintLimitExceed();
156         }
157         minterToMintedAmount[from_] = totalMintedNew;
158         uint256 availableLimit = minterToCap[from_] - totalMintedNew;
159         emit MintLimitDecreased(from_, availableLimit);
```

```
    }
```

Listing 3.1: `TokenController::_isEligibleToMint()`

Recommendation Revise the above `_isEligibleToMint()` helper for improved gas efficiency.

Status This issue has been fixed in the following commit: `281dd0c`.

3.2 Inconsistent NatSpec Comments in BaseXVSProxyOFT

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BaseXVSProxyOFT`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

The multichain support abstracts the `Layer Zero` functionalities in `BaseXVSProxyOFT` for seamless integration. In this section, we examine this base contract and notice certain `NatSpec` comments may be improved.

Specifically, we show below the code snippet from two related routines: `_isEligibleToSend()` and `_isEligibleToReceive()`. The former routine checks against the policies before the tokens are bridged out and the latter performs the same before the tokens are bridged in. However, it comes to our attention that the `NatSpec` comment in the first routine indicates `Check if the recipient's address is whitelisted` (line 231), which might be revised as `Check if the sender's address is whitelisted`. Similarly, the `NatSpec` comment in the latter routine shows `Check if the sender's address is whitelisted` (line 231), which might be revised as `Check if the recipient's address is whitelisted`.

```
230     function _isEligibleToSend(address from_, uint16 dstChainId_, uint256 amount_)
            internal {
231         // Check if the recipient's address is whitelisted
232         bool isWhiteListedUser = whitelist[from_];
233         ...
234     }
```

Listing 3.2: `BaseXVSProxyOFT::_isEligibleToSend()`

```
264     function _isEligibleToReceive(address toAddress_, uint16 srcChainId_, uint256
            receivedAmount_) internal {
265         // Check if the sender's address is whitelisted
266         bool isWhiteListedUser = whitelist[toAddress_];
267         ...
```

```
    }
```

Listing 3.3: `BaseXVSProxyOFT::_isEligibleToReceive()`

Recommendation Revise the above-mentioned routines with improved `NatSpec` comments.

Status This issue has been fixed in the following commit: `360228c`.

3.3 Removal of payable Modifier in XVSBridgeAdmin

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `XVSBridgeAdmin`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

To efficiently manage the multichain support, there is a dedicated `XVSBridgeAdmin` contract, which extends from `AccessControlledV8` for access control and manages the intended `XVSBridge`. Note that it maintains a registry of function signatures and names so that it can allow for dynamic function handling, i.e., checking of allowed interaction with intended functions. Our analysis shows the built-in `fallback` routine can be improved.

In the following, we show the implementation of its `fallback` routine. This routine will be invoked when the called function does not exist in the contract so that the inherited access control manager may kick in for the validation. It comes to our attention that this `fallback` routine has the `payable` modifier, which is not needed and can be safely removed.

```
    fallback(bytes calldata data) external payable returns (bytes memory) {
        string memory fun = _getFunctionName(msg.sig);
        require(bytes(fun).length != 0, "Function not found");
        _checkAccessAllowed(fun);
        (bool ok, bytes memory res) = address(XVSBridge).call(data);
        require(ok, "call failed");
        return res;
    }
```

Listing 3.4: `XVSBridgeAdmin::fallback()`

Recommendation Remove the `payable` modifier in the above `fallback` routine.

Status This issue has been fixed in the following commit: `da84da2`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the new multichain support, there is a privileged `owner` as well as the associated access control module, i.e., `AccessControlledV8`. They play a critical role in governing and regulating the system-wide operations (e.g., parameter setting and role authorization). They may also affect the flow of assets managed by this protocol. Our analysis shows that these privileged accounts need to be scrutinized. In the following, we examine the privileged accounts and their related privileged accesses in current contracts.

```
    function setOracle(address oracleAddress_) external onlyOwner {
        ensureNonzeroAddress(oracleAddress_);
        emit OracleChanged(address(oracle), oracleAddress_);
        oracle = ResilientOracleInterface(oracleAddress_);
    }

    /**
     * @notice Sets the limit of single transaction amount.
     * @param chainId_ Destination chain id.
     * @param limit_ Amount in USD(scaled with 18 decimals).
     * @custom:access Only owner.
     * @custom:event Emits SetMaxSingleTransactionLimit with old and new limit associated with chain id.
     */
    function setMaxSingleTransactionLimit(uint16 chainId_, uint256 limit_) external onlyOwner {
        emit SetMaxSingleTransactionLimit(chainId_, chainIdToMaxSingleTransactionLimit[chainId_], limit_);
        chainIdToMaxSingleTransactionLimit[chainId_] = limit_;
    }

    /**
     * @notice Sets the limit of daily (24 Hour) transactions amount.
     * @param chainId_ Destination chain id.
     * @param limit_ Amount in USD(scaled with 18 decimals).
     * @custom:access Only owner.
     * @custom:event Emits setMaxDailyLimit with old and new limit associated with chain id.
     */
    function setMaxDailyLimit(uint16 chainId_, uint256 limit_) external onlyOwner {
```

```
        require(limit_ >= chainIdToMaxSingleTransactionLimit[chainId_], "Daily limit <
            single transaction limit");
        emit SetMaxDailyLimit(chainId_, chainIdToMaxDailyLimit[chainId_], limit_);
        chainIdToMaxDailyLimit[chainId_] = limit_;
    }

    /**
     * @notice Sets the maximum limit for a single receive transaction.
     * @param chainId_ The destination chain ID.
     * @param limit_ The new maximum limit in USD(scaled with 18 decimals).
     * @custom:access Only owner.
     * @custom:event Emits setMaxSingleReceiveTransactionLimit with old and new limit
         associated with chain id.
     */
    function setMaxSingleReceiveTransactionLimit(uint16 chainId_, uint256 limit_)
        external onlyOwner {
        emit SetMaxSingleReceiveTransactionLimit(chainId_,
            chainIdToMaxSingleReceiveTransactionLimit[chainId_], limit_);
        chainIdToMaxSingleReceiveTransactionLimit[chainId_] = limit_;
    }
```

Listing 3.5: Example Privileged Operations in the `BaseXVSProxyOFT` 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. Note the owner of the bridge contracts (`XVSProxyOFTSrc`, `XVSProxyOFTDest`) will be an instance of the contract `XVSBridgeAdmin`. The owner of `XVSBridgeAdmin` will be the `Normal Timelock` contract. The privilege functions in the proxy contract will be executable only by the `Normal`, `Fast-track and Critical Timelock` contracts. These permissions will be granted via the `AccessControlManager` contract.

4 Conclusion

In this audit, we have analyzed the design and implementation of the multichain support in `Venus` with the goal of expanding into other blockchains, including `Ethereum mainnet`, `Arbitrum`, `Polygon zvEVM`, and `opBNB`, while still maintaining seamless user experience. 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 CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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