SMART CONTRACT AUDIT REPORT

for

Muquant Vault

Prepared By: Xiaomi Huang

PeckShield

August 18, 2023

Document Properties

Client	Muquant
Title	Smart Contract Audit Report
Target	Muquant Vault
Version	1.0
Author	Patrick Lou
Auditors	Xuxian Jiang, Patrick Lou
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 18, 2023	Patrick Lou	Final Release

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 Muquant Vault contract, 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 Muquant Vault

The Muquant Vault smart contract functions as a normal vault that supports asset deposit and withdrawal as well as other admin-related logic. Specifically, it allows users to deposit funds and later withdraw. The basic information of the audited contract is as follows:

Table 1.1: Basic Information of Muquant Vault

Item	Description
Target	Muquant Vault
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	August 18, 2023

In the following, we show the MD5 hash value of the file used in this audit:

- MD5 (Vault.sol) = 38a73756d9405e8a050062540c5d2acb

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.

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

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

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 `Muquant Vault` smart 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	
High	0	
Medium	0	
Low	1	🟦
Undetermined	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 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, the smart contract is 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 undetermined issue.

Table 2.1: Key Muquant Vault Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Undetermined	Revisited Logic of deposit()	Coding Practices	Confirmed
PVE-002	Low	Trust Issue of Admin Keys	Security Features	Confirmed

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 Revisited Logic of deposit()

- ID: PVE-001
- Severity: Undetermined
- Likelihood: High
- Impact: Low
- Target: Vault
- Category: Coding Practices [4]
- CWE subcategory: CWE-1109 [1]

Description

The Vault contract functions as a normal vault that supports asset deposit and withdrawal. While examining the deposit logic, we notice the need to revisit the current implementation.

To elaborate, we show below the implementation of the deposit() routine. It has a rather straightforward logic in recording the input token amount. However, it comes to our attention that the actual funds are not transferred into the vault. Meanwhile, in the related withdraw() counterpart routine, the requested funds are actually transferred to the intended recipient address (line 36). This raise the question on the need of actually transferring the funds into the vault in the first place.

```
23      function deposit(address token, address stable, uint256 amount) external {
24          require(keeper[msg.sender], "Vault: not keeper");
25          if (vaults[token].stable == address(0)) {
26              vaults[token].stable = stable;
27          }
28          vaults[token].amount += amount;
29      }
30
31      function withdraw(address token, address to, uint256 amount) external {
32          require(keeper[msg.sender], "Vault: not keeper");
33          require(vaults[token].amount >= amount, "Vault: not enough");
34          require(vaults[token].stable != address(0), "Vault: not exist");
35          vaults[token].amount -= amount;
36          SafeERC20.safeTransfer(IERC20(vaults[token].stable), to, amount);
37      }
```

Listing 3.1: Valut::deposit()/withdraw()

Recommendation Tranfer the funds into the vault in the above `deposit()`.

Status This issue has been confirmed as part of the design. Specifically, it is expected to transfer into the vault before calling the `deposit()` function.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Vault`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the `Vault` smart contract, there is a privileged `Guard` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., disable a specific `keeper` from withdrawing asset). In the following, we show the representative functions potentially affected by the privilege of the `Guard` account.

```
    function withdraw(address token, address to, uint256 amount) external {
        require(keeper[msg.sender], "Vault: not keeper");
        require(vaults[token].amount >= amount, "Vault: not enough");
        require(vaults[token].stable != address(0), "Vault: not exist");
        vaults[token].amount -= amount;
        SafeERC20.safeTransfer(IERC20(vaults[token].stable), to, amount);
    }

    function safe(address _keeper) external {
        require(msg.sender == Guard, "Vault: not guard");
        require(mkeeper[_keeper], "Vault: not keeper");
        keeper[_keeper] = !keeper[_keeper];
    }
```

Listing 3.2: Example Privileged Operations in `Valut`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the `Guard` is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 The issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Muquant Vault` smart contract, which functions as a normal vault. It supports asset deposit and withdrawal as well as other admin-related logic. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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.