SMART CONTRACT AUDIT REPORT

for

DODOMININGV2/DODOSTABLEPOOL

Prepared By: Shuxiao Wang

PeckShield

April 10, 2021

Document Properties

Client	DODO
Title	Smart Contract Audit Report
Target	DODOMiningV2 + DODOStablePool
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	April 10, 2021	Xuxian Jiang	Release Candidate
0.1	April 8, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of **DODOMiningV2** and **DODOStablePool**, 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 DODOMiningV2 And DODOStablePool

DODO is an innovative, next-generation on-chain liquidity provision solution. It recognizes main drawbacks of current AMM algorithms (especially in provisioning unstable portfolios and having relatively low funding utilization rates), and accordingly proposes an algorithm that imitates human market makers to bring sufficient on-chain liquidity. The audited system includes `DODOMiningV2`, which supports various pool tokens, ERC20-based tokens, as well as `vDODO`. One unique aspect of `DODO Mining V2` is the support of allowing users to mine more than one reward tokens at the same time. In addition, the audited system includes `DODOStablePool (DSP)`, a new type of pool that adopts the innovative PMM algorithm for specialized public pools with stable trading pairs.

The basic information of DODOMiningV2 is as follows:

Table 1.1: Basic Information of DODOMiningV2

Item	Description
Issuer	DODO
Website	https://app.dododex.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 10, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. As mentioned earlier, DODOMiningV2 assumes a trusted oracle with timely market price feeds and the oracle itself is not part of this audit.

- https://github.com/DODOEX/contractV2.git (5917c95)

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

- https://github.com/DODOEX/contractV2.git (087f7b2)

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 accordingly classified into four categories, 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 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 1.3: 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

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 DODOMiningV2/DODOStablePool design and implementation. 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	0
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 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Addition of rescueToken() to RewardVault	Coding Practices	Fixed
PVE-002	Low	Improved Logic In withdrawLeftOver()	Business Logic	Fixed

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 Suggested Addition of rescueToken() to RewardVault

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: RewardVault
- Category: Coding Practices [3]
- CWE subcategory: CWE-1099 [1]

Description

By design, the DODOMineV2 support has different RewardVault contracts that hold various types of reward assets. From past experience with current popular DeFi protocols, e.g., YFI/Curve, we notice that there is always non-trivial possibilities that non-related tokens may be accidentally sent to the pool contract(s). Moreover, the DODOMineV2 support allows for the dynamic adjustment of rewardPerBlock and endBlock, which inevitably introduces the possibility of having leftover amount in RewardVault contracts. To avoid unnecessary loss of protocol users, we suggest to add the support of rescuing remaining reward tokens. This is a design choice for the benefit of protocol users.

Recommendation Add the support of rescuing remaining tokens in RewardVault. An example addition is shown below:

```
function recoverERC20(address _token, uint256 _amount) external onlyOwner {
    IERC20(_token).safeTransfer(owner(), _amount);
    emit Recovered(_token, _amount);
}
```

Listing 3.1: RewardVault::recoverERC20()

Status This issue has been fixed in the following commit: d415ce8.

3.2 Improved Logic In withdrawLeftOver()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseMine`, `RewardVault`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

As mentioned in Section 3.1, the new mining support allows for dynamic adjustment of `rewardPerBlock` and `endBlock`, and therefore inevitably introduces the possibility of having leftover amount in `RewardVault` contracts. To mitigate, the team has designed a permissioned function `withdrawLeftOver()` that can be exercised to withdraw remaining assets after the rewarding time period expires.

To elaborate, we show below the `withdrawLeftOver()` function in both `BaseMine` and `RewardVault` contracts. It indeed achieves the intended purpose. However, it comes to our attention that the current implementation is somehow programmed to be able to withdraw all remaining funds as far as the rewarding time period expires, i.e., `require(block.number > rt.endBlock)` (line 188). This may impose unnecessary restriction on the participating users to make their reward claims immediately before the end of reward session. Otherwise, they run into the risk of losing their rewarded tokens.

```
186     function withdrawLeftOver(uint256 i) external onlyOwner {
187         RewardTokenInfo storage rt = rewardTokenInfos[i];
188         require(block.number > rt.endBlock, "DODOMineV2: MINING_NOT_FINISHED");

190         IRewardVault(rt.rewardVault).withdrawLeftOver(msg.sender);

192         emit WithdrawLeftOver(msg.sender, i);
193     }
```

Listing 3.2: BaseMine::withdrawLeftOver()

```
    function withdrawLeftOver(address to) external onlyOwner {
        uint256 leftover = IERC20(rewardToken).balanceOf(address(this));
        IERC20(rewardToken).safeTransfer(to, leftover);
    }
```

Listing 3.3: RewardVault::withdrawLeftOver()

Recommendation Revisit the `withdrawLeftOver()` logic to avoid withdrawing all reward tokens since not all mining users are able to immediately make claims before `endBlock`.

Status This issue has been fixed in the following commit: `087f7b2`.

4 | Conclusion

In this audit, we have analyzed the documentation and implementation of `DODOMineV2 and DODOStablePool`. The audited system presents a unique innovation and we are impressed by the overall design and solid implementation. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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