SMART CONTRACT AUDIT REPORT

for

DODO MINEUPDATE

Prepared By: Yiqun Chen

PeckShield

July 2, 2021

Document Properties

Client	DODO
Title	Smart Contract Audit Report
Target	DODO MineUpdate
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 2, 2021	Xuxian Jiang	Final Release

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

1 Introduction

Given the opportunity to review the design document and related smart contract source code of **DODO MineUpdate**, 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 DODO MineUpdate

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 the `MineUpdate` upgrade, which improves earlier versions of mining feature as well as the associated factories and proxies.

The basic information of DODO MineUpdate is as follows:

Table 1.1: Basic Information of DODO MineUpdate

Item	Description
Issuer	DODO
Website	https://app.dododex.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 2, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the repository has a number of subdirectories and this audit mainly focuses on the `DODOToken/DODOMineV3` subdirectory.

- https://github.com/DODOEX/contractV2.git (c7202ee)

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 (0bdc6f5)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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) [9], 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 DODO MineUpdate 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	3	■■■
Informational	1	■
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 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 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	ERC20 Compliance Of CustomERC20	Coding Practices	Fixed
PVE-002	Low	Improved Reentrancy Prevention in ERC20Mine	Time and State	Fixed
PVE-003	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Fixed
PVE-004	Low	Accommodation of Possible Non-ERC20-Compliance	Numeric Errors	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 ERC20 Compliance Of CustomERC20

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CustomERC20`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1099 [1]

Description

The `DODOMineV3` support has a customized `ERC20` token contract implementation in `CustomERC20`. Our analysis with this specific token contract reveals a minor ERC20-compliance issue.

To elaborate, we show below the main storage state variables. We notice that the state `decimals` is defined as `uint256` (line 17). For better compliance with the ERC20 standard, it needs to be defined as `uint8`.

```
13 contract CustomERC20 is InitializableOwnable {
14     using SafeMath for uint256;
15
16     string public name;
17     uint256 public decimals;
18     string public symbol;
19     uint256 public totalSupply;
20
21     uint256 public tradeBurnRatio;
22     uint256 public tradeFeeRatio;
23     address public team;
24     bool public isMintable;
25     ...
26 }
```

Listing 3.1: The `CustomERC20` Contract

Recommendation Revise the `decimals` state to have the `uint8` type, not `uint256`.

Status This issue has been fixed in the following commit: `7acd630`.

3.2 Improved Reentrancy Prevention in ERC20Mine

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ERC20Mine`
- Category: Time and State [7]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. In particular, within the `ERC20Mine` contract, the `deposit()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 41) starts before effecting the update on internal states (lines 44 – 45), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `deposit()` function.

```
35    function deposit(uint256 amount) external {
36        require(amount > 0, "DODOMineV3: CANNOT_DEPOSIT_ZERO");
37
38        _updateAllReward(msg.sender);
39
40        uint256 erc20OriginBalance = IERC20(_TOKEN_).balanceOf(address(this));
41        IERC20(_TOKEN_).safeTransferFrom(msg.sender, address(this), amount);
42        uint256 actualStakeAmount = IERC20(_TOKEN_).balanceOf(address(this)).sub(
              erc20OriginBalance);
43
44        _totalSupply = _totalSupply.add(actualStakeAmount);
45        _balances[msg.sender] = _balances[msg.sender].add(actualStakeAmount);
46
47        emit Deposit(msg.sender, actualStakeAmount);
```

```
    }
```

Listing 3.2: `ERC20Mine::deposit()`

Recommendation Apply necessary `re-entrancy` prevention to the above `deposit()` function.

Status This issue has been fixed in the following commit: `7acd630`.

3.3 Improved Precision By Multiplication And Division Reordering

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `DODORouteProxy`
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `DODORouteProxy::_multiSwap()` as an example. This routine is designed to perform multiple token swaps.

```
    function _multiSwap(
        uint256[] memory totalWeight,
        address[] memory midToken,
        uint256[] memory splitNumber,
        bytes[] memory swapSequence,
        address[] memory assetFrom
    ) internal {
        for(uint256 i = 1; i < splitNumber.length; i++) {
            // define midtoken address, ETH -> WETH address
            uint256 curTotalAmount = IERC20(midToken[i]).tokenBalanceOf(assetFrom[i-1]);
            uint256 curTotalWeight = totalWeight[i-1];

            for(uint256 j = splitNumber[i-1]; j < splitNumber[i]; j++) {
                PoolInfo memory curPoolInfo;
                {
```

```
186                     (address pool, address adapter, uint256 mixPara, bytes memory
                            moreInfo) = abi.decode(swapSequence[j], (address, address,
                            uint256, bytes));

188                     curPoolInfo.direction = mixPara >> 17;
189                     curPoolInfo.weight = (0xffff & mixPara) >> 9;
190                     curPoolInfo.poolEdition = (0xff & mixPara);
191                     curPoolInfo.pool = pool;
192                     curPoolInfo.adapter = adapter;
193                     curPoolInfo.moreInfo = moreInfo;
194                 }

196                 if(assetFrom[i-1] == address(this)) {
197                     uint256 curAmount = curTotalAmount.div(curTotalWeight).mul(
                            curPoolInfo.weight);

199                     if(curPoolInfo.poolEdition == 1) {
200                         //For using transferFrom pool (like dodoV1, Curve)
201                         IERC20(midToken[i]).transfer(curPoolInfo.adapter, curAmount);
202                     } else {
203                         //For using transfer pool (like dodoV2)
204                         IERC20(midToken[i]).transfer(curPoolInfo.pool, curAmount);
205                     }
206                 }

208                 if(curPoolInfo.direction == 0) {
209                     IDODOAdapter(curPoolInfo.adapter).sellBase(assetFrom[i], curPoolInfo
                            .pool, curPoolInfo.moreInfo);
210                 } else {
211                     IDODOAdapter(curPoolInfo.adapter).sellQuote(assetFrom[i],
                            curPoolInfo.pool, curPoolInfo.moreInfo);
212                 }
213             }
214         }
215     }
```

Listing 3.3: `DODORouteProxy::_multiSwap()`

We notice the calculation of the internal `curAmount` (line 1197) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `curAmount = curTotalAmount.mul(curPoolInfo.weight).div(curTotalWeight)`. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in the following commit: `7acd630`.

3.4 Accommodation of Possible Non-ERC20-Compliance

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseMine`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transferFrom()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transferFrom()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transferFrom()` interface with a `bool` return value. As a result, the call to `transferFrom()` may expect a return value. With the lack of return value of `USDT`'s `transferFrom()`, the call will be unfortunately reverted.

```
    function transferFrom(address _from, address _to, uint _value) public
        onlyPayloadSize(3 * 32) {
        var _allowance = allowed[_from][msg.sender];

        // Check is not needed because sub(_allowance, _value) will already throw if
            this condition is not met
        // if (_value > _allowance) throw;

        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        if (_allowance < MAX_UINT) {
            allowed[_from][msg.sender] = _allowance.sub(_value);
        }
        uint sendAmount = _value.sub(fee);
        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(_from, owner, fee);
        }
        Transfer(_from, _to, sendAmount);
    }
```

Listing 3.4: USDT Token Contract

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transfer()` as well, i.e., `safeApprove()/safeTransfer()`.

In current implementation, if we examine the `BaseMine::addRewardToken()` routine that is designed to add a new reward token. To accommodate the specific idiosyncrasy, there is a need to use `safeTransfer()`, instead of `transfer()` (line 176).

```
149     function addRewardToken(
150         address rewardToken,
151         uint256 rewardPerBlock,
152         uint256 startBlock,
153         uint256 endBlock
154     ) external onlyOwner {
155         require(rewardToken != address(0), "DODOMineV3: TOKEN_INVALID");
156         require(startBlock > block.number, "DODOMineV3: START_BLOCK_INVALID");
157         require(endBlock > startBlock, "DODOMineV3: DURATION_INVALID");

159         uint256 len = rewardTokenInfos.length;
160         for (uint256 i = 0; i < len; i++) {
161             require(
162                 rewardToken != rewardTokenInfos[i].rewardToken,
163                 "DODOMineV3: TOKEN_ALREADY_ADDED"
164             );
165         }

167         RewardTokenInfo storage rt = rewardTokenInfos.push();
168         rt.rewardToken = rewardToken;
169         rt.startBlock = startBlock;
170         rt.lastFlagBlock = startBlock;
171         rt.endBlock = endBlock;
172         rt.rewardPerBlock = rewardPerBlock;
173         rt.rewardVault = address(new RewardVault(rewardToken));

175         uint256 rewardAmount = rewardPerBlock.mul(endBlock.sub(startBlock));
176         IERC20(rewardToken).transfer(rt.rewardVault, rewardAmount);
177         RewardVault(rt.rewardVault).syncValue();

179         emit NewRewardToken(len, rewardToken);
180     }
```

Listing 3.5: `BaseMine::addRewardToken()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been fixed in the following commit: `7acd630`.

4 Conclusion

In this audit, we have analyzed the documentation and implementation of `DODO MineUpdate`. 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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 CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.