



SMART CONTRACT AUDIT REPORT

for

DODO



Prepared By: Shuxiao Wang

Hangzhou, China

July 10, 2020

Document Properties

Client	DODO
Title	Smart Contract Audit Report
Target	DODO
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Xuxian Jiang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 10, 2020	Xuxian Jiang	Final Release
1.0-rc1	July 8, 2020	Xuxian Jiang	Additional Findings #3
0.2	July 2, 2020	Xuxian Jiang	Additional Findings #2
0.1	June 30, 2020	Xuxian Jiang	First Release #1

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	5
1.1	About DODO	5
1.2	About PeckShield	6
1.3	Methodology	6
1.4	Disclaimer	8
2	Findings	10
2.1	Summary	10
2.2	Key Findings	11
3	Detailed Results	12
3.1	Non-ERC20-Compliant DODOLpTokens	12
3.2	Improved Precision Calculation in DODOMath	13
3.3	Improved Precision Calculation #2 in DODOMath	15
3.4	approve()/transferFrom() Race Condition	17
3.5	Better Handling of Privilege Transfers	18
3.6	Centralized Governance	20
3.7	Possible Integer Overflow in <i>sqrt()</i>	21
3.8	Redundant State Checks	22
3.9	Contract Verification in breedDODO()	23
3.10	Balance Inconsistency With Deflationary Tokens	25
3.11	Aggregated Transfer of Maintainer Fees	26
3.12	Misleading Embedded Code Comments	28
3.13	Missing DODO Validation in DODOEthProxy	29
3.14	Other Suggestions	30
4	Conclusion	32
5	Appendix	33
5.1	Basic Coding Bugs	33

5.1.1	Constructor Mismatch	33
5.1.2	Ownership Takeover	33
5.1.3	Redundant Fallback Function	33
5.1.4	Overflows & Underflows	33
5.1.5	Reentrancy	34
5.1.6	Money-Giving Bug	34
5.1.7	Blackhole	34
5.1.8	Unauthorized Self-Destruct	34
5.1.9	Revert DoS	34
5.1.10	Unchecked External Call	35
5.1.11	Gasless Send	35
5.1.12	Send Instead Of Transfer	35
5.1.13	Costly Loop	35
5.1.14	(Unsafe) Use Of Untrusted Libraries	35
5.1.15	(Unsafe) Use Of Predictable Variables	36
5.1.16	Transaction Ordering Dependence	36
5.1.17	Deprecated Uses	36
5.2	Semantic Consistency Checks	36
5.3	Additional Recommendations	36
5.3.1	Avoid Use of Variadic Byte Array	36
5.3.2	Make Visibility Level Explicit	37
5.3.3	Make Type Inference Explicit	37
5.3.4	Adhere To Function Declaration Strictly	37
References		38

1 | Introduction

Given the opportunity to review the **DODO** design document and related smart contract source code, we in the report outline 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

DODO is an innovative, next-generation on-chain liquidity provision solution. It is purely driven by a so-called *Proactive Market Market* or PMM algorithm that outperforms current popular *Automatic Market Maker* of AMM algorithms. In particular, 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. Assuming a timely market price feed, the algorithm proactively adjusts trading prices around the feed, hence better providing on-chain liquidity and protecting liquidity providers' portfolios (by avoiding unnecessary loss to arbitrageurs). DODO advances the current DEX frontline and is considered an indeed innovation in the rapidly-evolving DeFi ecosystem.

The basic information of DODO is as follows:

Table 1.1: Basic Information of DODO

Item	Description
Issuer	DODO
Website	https://github.com/radar-bear/dodo-smart-contract
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 10, 2020

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

- <https://github.com/radar-bear/dodo-smart-contract> (d749640)

1.2 About PeckShield

PeckShield Inc. [21] 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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [16]:

- 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) [15], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit 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 an 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 DODO implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	6	
Informational	5	
Total	13	

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 medium-severity vulnerability, 6 low-severity vulnerabilities, and 5 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Non-ERC20-Compliant DODOLpTokens	Coding Practices	Fixed
PVE-002	Low	Improved Precision Calculation in DODOMath	Numeric Errors	Fixed
PVE-003	Low	Improved Precision Calculation #2 in DODOMath	Numeric Errors	Fixed
PVE-004	Low	approve()/transferFrom() Race Condition	Time and State	Confirmed
PVE-005	Info.	Better Handling of Ownership Transfer	Security Features	Fixed
PVE-006	Info.	Centralized Governance	Security Features	Confirmed
PVE-007	Low	Possible Integer Overflow in sqrt()	Numeric Errors	Fixed
PVE-008	Info.	Redundant State Checks	Security Features	Confirmed
PVE-009	Info.	Contract Verification in breedDODO()	Security Features	Confirmed
PVE-010	Medium	Balance Inconsistency With Deflationary Tokens	Time and State	Fixed
PVE-011	Low	Aggregated Transfer of Maintainer Fees	Time and State	Confirmed
PVE-012	Info.	Misleading Embedded Code Comments	Coding Practices	Fixed
PVE-013	Medium	Missing DODO Validation in DODOEthProxy	Coding Practices	Fixed

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Non-ERC20-Compliant DODOLpTokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DODOLpToken
- Category: Coding Practices [13]
- CWE subcategory: CWE-1099 [3]

Description

To tokenize the assets in a liquidity pool and reflect a liquidity provider's share in the pool, DODO provides DODOLpToken, an ERC20-based contract with additional functionalities for managed `mint` and `burn`. Specifically, the DODOLpToken tokens will be accordingly issued to (or burned from) a liquidity provider upon each deposit (or withdrawal).

```
20 contract DODOLpToken is Ownable {  
21     using SafeMath for uint256;  
22  
23     uint256 public totalSupply;  
24     mapping(address => uint256) internal balances;  
25     mapping(address => mapping(address => uint256)) internal allowed;  
26  
27     ...  
28 }
```

Listing 3.1: contracts/impl/DODOLpToken.sol

We notice that current implementation of DODOLpToken lacks a few standard fields, i.e., `name`, `symbol` and `decimals`. The lack of these standard fields leads to non-compliance of ERC20 and may cause unnecessary confusions in naming, inspecting, and transferring these tokenized share. An improvement is possible by either initializing these fields as hard-coded constants in the DODOLpToken contract or directly associated with the respective `baseToken`/`quoteToken`. Note all token-pairs or exchanges in DODO might share the same name and symbol – a similar approach has been taken

by `Uniswap` and no major issues have been observed in the wild. However, it is better to have unique `name` or `symbol` that can be associated with the pairing `baseToken` and `quoteToken`.

In addition, for the `burn()` routine, there is a `burn` event to indicate the reduction of `totalSupply`. It is also meaningful to generate a second event that indicates a `transfer` to 0 address.

```

113     function burn(address user, uint256 value) external onlyOwner {
114         balances[user] = balances[user].sub(value);
115         totalSupply = totalSupply.sub(value);
116         emit Burn(user, value);
117         emit Transfer(user, address(0), value); // <-- Currently missing
118     }

```

Listing 3.2: `contracts/impl/DODOLpToken.sol`

Recommendation It is strongly suggested to follow the standard ERC20 convention and define missing fields, including `name`, `symbol` and `decimals`.

3.2 Improved Precision Calculation in DODOMath

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `DODOMath`
- Category: Numeric Errors [14]
- CWE subcategory: CWE-190 [5]

Description

According to the DODO's PMM algorithm, its unique price curve is continuous but with two distinct segments and three different operating states: `R0ne`, `RAbove`, and `RBelow`. The first state `R0ne` reflects the expected state of being balanced between `baseToken` and `quoteToken` assets and its trading price is well aligned with current market price; The second state `RAbove` reflects the state of having more balance of `quoteToken` than expected and there is a need to attempt to sell more `quoteToken` to bring the state back to `R0ne`; The third state `RBelow` on the contrary reflects the state of having more balance of `baseToken` than expected and there is a need to attempt to sell more `baseToken` to bring the state back to `R0ne`.

The transition among these three states is triggered by users' trading behavior (especially the trading amount) and also affected by real-time market price feed. Naturally, the transition requires complex computation (implemented in `DODOMatch`). In particular, `DODOMatch` has three operations: one specific integration and two other quadratic solutions. The integration computation, i.e., `_GeneralIntegrate()`, is used in `R0ne` and `RAbove` to calculate the expected exchange of `quoteToken` for the trading `baseToken` amount. The quadratic solution `_SolveQuadraticFunctionForTrade()` is used in

R0ne and RBelow for the very same purpose. Another quadratic solution `_SolveQuadraticFunctionForTarget()` is instead used in RAbove and RBelow to calculate required token-pair amounts if we want to bring the state back to R0ne.

Note that the lack of `float` support in [Solidity](#) makes the above calculations unusually complicated. And `_GeneralIntegrate()` can be further improved as current calculation may lead to possible precision loss. Specifically, as shown in lines 38–39 (see the code snippet below), `V0V1 = DecimalMath.divCeil(V0, V1)`; and `V0V2 = DecimalMath.divCeil(V0, V2)`.

```

30     function _GeneralIntegrate(
31         uint256 V0,
32         uint256 V1,
33         uint256 V2,
34         uint256 i,
35         uint256 k
36     ) internal pure returns (uint256) {
37         uint256 fairAmount = DecimalMath.mul(i, V1.sub(V2)); // i*delta
38         uint256 V0V1 = DecimalMath.divCeil(V0, V1); // V0/V1
39         uint256 V0V2 = DecimalMath.divCeil(V0, V2); // V0/V2
40         uint256 penalty = DecimalMath.mul(DecimalMath.mul(k, V0V1), V0V2); // k(V0^2/V1/
           V2)
41         return DecimalMath.mul(fairAmount, DecimalMath.ONE.sub(k).add(penalty));
42     }

```

Listing 3.3: `contracts/impl/DODOMath.sol`

For improved precision, it is better to calculate the multiplication before the division, i.e., `V0V0V1V2 = DecimalMath.divCeil(DecimalMath.divCeil(DecimalMath.mul(V0, V0), V1), V2)`;

There is another similar issue in the calculation of `_SolveQuadraticFunctionForTarget()` (line 111) that can also benefit from the above calculation with improved precision: `uint256 sqrt = DecimalMath.divCeil(DecimalMath.mul(k, fairAmount).mul(4), V1)`. Note if there is a rounding issue, it is preferable to allow the calculation lean towards the liquidity pool to ensure DODO is balanced. Therefore, our calculation also replaces `divFloor` with `divCeil` for the very same reason.

```

105     function _SolveQuadraticFunctionForTarget(
106         uint256 V1,
107         uint256 k,
108         uint256 fairAmount
109     ) internal pure returns (uint256 V0) {
110         // V0 = V1+V1*(sqrt-1)/2k
111         uint256 sqrt = DecimalMath.divFloor(DecimalMath.mul(k, fairAmount), V1).mul(4);
112         sqrt = sqrt.add(DecimalMath.ONE).mul(DecimalMath.ONE).sqrt();
113         uint256 premium = DecimalMath.divFloor(sqrt.sub(DecimalMath.ONE), k.mul(2));
114         // V0 is greater than or equal to V1 according to the solution
115         return DecimalMath.mul(V1, DecimalMath.ONE.add(premium));
116     }

```

Listing 3.4: `contracts/impl/DODOMath.sol`

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

3.3 Improved Precision Calculation #2 in DODOMath

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: DODOMath
- Category: Numeric Errors [14]
- CWE subcategory: CWE-190 [5]

Description

In previous Section 3.2, we examine a particular precision issue. In this section, we examine another related precision issue. As mentioned earlier, the DODO's PMM algorithm operates in its continuous price curve with two distinct segments and three different states (aka scenarios): `R0ne`, `RAbove`, and `RBelow`. Accordingly, the `DODOMath` contract implements three operations: one specific integration (`_GeneralIntegrate()`) and two other quadratic solutions (`_SolveQuadraticFunctionForTrade()` and `_SolveQuadraticFunctionForTrade()`).

```

58     function _SolveQuadraticFunctionForTrade(
59         uint256 Q0,
60         uint256 Q1,
61         uint256 ideltaB ,
62         bool deltaBSig ,
63         uint256 k
64     ) internal pure returns (uint256) {
65         // calculate -b value and sig
66         // -b = (1-k)Q1-kQ0^2/Q1+i*deltaB
67         uint256 kQ02Q1 = DecimalMath.mul(k, Q0).mul(Q0).div(Q1); // kQ0^2/Q1
68         uint256 b = DecimalMath.mul(DecimalMath.ONE.sub(k), Q1); // (1-k)Q1
69         bool minusbSig = true;
70         if (deltaBSig) {
71             b = b.add(ideltaB); // (1-k)Q1+i*deltaB
72         } else {
73             kQ02Q1 = kQ02Q1.add(ideltaB); // -i*(-deltaB)-kQ0^2/Q1
74         }
75         if (b >= kQ02Q1) {
76             b = b.sub(kQ02Q1);
77             minusbSig = true;
78         } else {
79             b = kQ02Q1.sub(b);
80             minusbSig = false;
81         }

82         // calculate sqrt
83         uint256 squareRoot = DecimalMath.mul(
84             DecimalMath.ONE.sub(k).mul(4) ,
85             DecimalMath.mul(k, Q0).mul(Q0)
86         ); // 4(1-k)kQ0^2
87         squareRoot = b.mul(b).add(squareRoot).sqrt(); // sqrt(b*b-4(1-k)kQ0*Q0)

```

```

90     // final res
91     uint256 denominator = DecimalMath.ONE.sub(k).mul(2); // 2(1-k)
92     if (minusbSig) {
93         return DecimalMath.divFloor(b.add(squareRoot), denominator);
94     } else {
95         return DecimalMath.divFloor(squareRoot.sub(b), denominator);
96     }

```

Listing 3.5: contracts/impl/DODOMath.sol

In this section, we examine the last quadratic solution. Specifically, this quadratic solution `_SolveQuadraticFunctionForTarget()` is used in `R0ne` and `RBelow` to calculate the expected exchange of `quoteToken` for the trading `baseToken` amount. The function has several arguments (lines 59 – 63) where the fourth one, i.e., `deltaBSig`, is used to indicate whether this particular trade will lead to increased or decreased balance of `quoteToken`. If `deltaBSig=true`, then the trading amount is calculated by `Q2.sub(quoteBalance)`; If `deltaBSig=false`, then the trading amount is calculated by `quoteBalance.sub(Q2)`. Note that `Q2` is the calculated result from the quadratic solution while `quoteBalance` is current `quoteToken` balance. Following the same principle as mentioned in Section 3.2, for precision-related calculations, it is preferable to lean the calculation towards the liquidity pool to ensure that DODO is always balanced. Therefore, our calculation here needs to replace the final `divFloor` with `divCeil` on the condition of `deltaBSig=true`.

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

```

58     function _SolveQuadraticFunctionForTrade(
59         uint256 Q0,
60         uint256 Q1,
61         uint256 ideltaB,
62         bool deltaBSig,
63         uint256 k
64     ) internal pure returns (uint256) {
65         // calculate -b value and sig
66         // -b = (1-k)Q1-kQ0^2/Q1+i*deltaB
67         uint256 kQ02Q1 = DecimalMath.mul(k, Q0).mul(Q0).div(Q1); // kQ0^2/Q1
68         uint256 b = DecimalMath.mul(DecimalMath.ONE.sub(k), Q1); // (1-k)Q1
69         bool minusbSig = true;
70         if (deltaBSig) {
71             b = b.add(ideltaB); // (1-k)Q1+i*deltaB
72         } else {
73             kQ02Q1 = kQ02Q1.add(ideltaB); // -i*(-deltaB)-kQ0^2/Q1
74         }
75         if (b >= kQ02Q1) {
76             b = b.sub(kQ02Q1);
77             minusbSig = true;
78         } else {
79             b = kQ02Q1.sub(b);
80             minusbSig = false;
81         }

```



```

83      // calculate sqrt
84      uint256 squareRoot = DecimalMath.mul(
85          DecimalMath.ONE.sub(k).mul(4),
86          DecimalMath.mul(k, Q0).mul(Q0)
87      ); // 4(1-k)kQ0^2
88      squareRoot = b.mul(b).add(squareRoot).sqrt(); // sqrt(b*b-4(1-k)kQ0*Q0)

90      // final res
91      uint256 denominator = DecimalMath.ONE.sub(k).mul(2); // 2(1-k)
92      if (minusbSig) {
93          numerator = b.add(squareRoot);
94      } else {
95          numerator = squareRoot.sub(b);
96      }

98      if (deltaBSig) {
99          return DecimalMath.divFloor(numerator, denominator);
100     } else {
101         return DecimalMath.divCeil(numerator, denominator);
102     }

```

Listing 3.6: contracts/impl/DODOMath.sol (revised)

3.4 approve()/transferFrom() Race Condition

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: DODOLpToken
- Category: Time and State [12]
- CWE subcategory: CWE-362 [8]

Description

DODOLpToken is an ERC20 token that represents the liquidity providers' shares on the tokenized pools. In current implementation, there is a known race condition issue regarding `approve()` / `transferFrom()` [2]. Specifically, when a user intends to reduce the allowed spending amount previously approved from, say, 10 DODO to 1 DODO. The previously approved spender might race to transfer the amount you initially approved (the 10 DODO) and then additionally spend the new amount you just approved (1 DODO). This breaks the user's intention of restricting the spender to the new amount, **not** the sum of old amount and new amount.

In order to properly approve tokens, there also exists a known workaround: users can utilize the `increaseApproval` and `decreaseApproval` non-ERC20 functions on the token versus the traditional `approve` function.

```

85  /**
86   * @dev Approve the passed address to spend the specified amount of tokens on behalf
      of msg.sender.
87   * @param spender The address which will spend the funds.
88   * @param amount The amount of tokens to be spent.
89   */
90   function approve(address spender, uint256 amount) public returns (bool) {
91       allowed[msg.sender][spender] = amount;
92       emit Approval(msg.sender, spender, amount);
93       return true;
94   }

```

Listing 3.7: contracts/impl/DODOLpToken.sol

Recommendation Add the suggested workaround functions `increaseApproval()/decreaseApproval()`. However, considering the difficulty and possible lean gains in exploiting the race condition, we also think it is reasonable to leave it as is.

3.5 Better Handling of Privilege Transfers

- ID: PVE-005
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Targets: Ownable, Admin
- Category: Security Features [11]
- CWE subcategory: CWE-282 [7]

Description

DODO implements a rather basic access control mechanism that allows a privileged account, i.e., `_OWNER_` or `_SUPERVISOR_`, to be granted exclusive access to typically sensitive functions (e.g., the setting of `_ORACLE_` and `_MAINTAINER_`). Because of the privileged access and the implications of these sensitive functions, the `_OWNER_` and `_SUPERVISOR_` accounts are essential for the protocol-level safety and operation. In the following, we elaborate with the `_OWNER_` account.

Within the governing contract `Ownable`, a specific function, i.e., `transferOwnership(address newOwner)`, is provided to allow for possible `_OWNER_` updates. However, current implementation achieves its goal within a single transaction. This is reasonable under the assumption that the `newOwner` parameter is always correctly provided. However, in the unlikely situation, when an incorrect `newOwner` is provided, the contract owner may be forever lost, which might be devastating for DODO operation and maintenance.

As a common best practice, instead of achieving the owner update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the owner update intent and the second step accepts and materializes the update. Both steps should be executed in two

separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract ownership to an uncontrolled address. In other words, this two-step procedure ensures that an owner public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the owner transfer process.

```

38     function transferOwnership(address newOwner) external onlyOwner {
39         require(newOwner != address(0), "INVALID_OWNER");
40         emit OwnershipTransferred(_OWNER_, newOwner);
41         _OWNER_ = newOwner;
42     }

```

Listing 3.8: lib/Ownabe.sol

Recommendation Implement a two-step approach for owner update (or transfer): `setOwner()` and `acceptOwner()`. The same is also applicable for other privileged accounts, i.e., `_SUPERVISOR_`. In addition, generate meaningful events (currently missing) whenever there is a privileged account transfer.

```

38     address public _newOwner;

40     /*
41      * Set new manager address.
42      */
43     function setOwner(
44         address newOwner
45     )
46     external
47     onlyOwner
48     {
49         require(newOwner != address(0), "setOwner: new owner is the zero address");
50         require(newOwner != _newOwner, "setOwner: new owner is the same as previous
           owner");

52         _newOwner = newOwner;
53     }

55     function acceptOwner() public {
56         require(msg.sender == _newOwner);

58         emit OwnershipTransferred(_OWNER_, _newOwner);

60         _OWNER_ = _newOwner;
61         _newOwner = 0x0;

63     }

```

Listing 3.9: lib/Ownabe.sol (revised)

3.6 Centralized Governance

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Targets: DODOZoo, DODO, Settlement
- Category: Security Features [11]
- CWE subcategory: CWE-654 [10]

Description

Throughout the whole DODO system, the `_OWNER_` is the account who can access or execute all the privileged functions (via the `onlyOwner` modifier). However, some privileged functions are not necessary to be controlled by the `_OWNER_` account. For example, the access to certain functions (e.g. `setLiquidityProviderFeeRate/setMaintainerFeeRate`) could be assigned to an operator/manager account or controlled by a multisig account.

The current centralized implementation makes this system not compatible to the usual setup towards community-oriented governance for shared responsibilities or reduced risks. It is understandable that the system intends to begin with a centralized governance in the early, formative days and then gradually shift over time to a community-oriented governance system. It will be greatly helpful to think ahead and materialize necessary plan to have a community-oriented governance, which could move the system one step further toward ultimate decentralization. Moreover, such governance mechanism might be naturally associated with a governance token whose distribution (or other incentive approaches) can be designed to engage the community by linking it together with the business logic in DODO.

The same concern is also applicable when a token pair needs to be settled. As shown in the below code snippet, the current `finalSettlement()` can be initiated only by current owner.

```

82 // last step to shut down dodo
83 function finalSettlement() external onlyOwner notClosed {
84     _CLOSED_ = true;
85     _DEPOSIT_QUOTE_ALLOWED_ = false;
86     _DEPOSIT_BASE_ALLOWED_ = false;
87     _TRADE_ALLOWED_ = false;
88     uint256 totalBaseCapital = getTotalBaseCapital();
89     uint256 totalQuoteCapital = getTotalQuoteCapital();

91     if (_QUOTE_BALANCE_ > _TARGET_QUOTE_TOKEN_AMOUNT_) {
92         uint256 spareQuote = _QUOTE_BALANCE_.sub(_TARGET_QUOTE_TOKEN_AMOUNT_);
93         _BASE_CAPITAL_RECEIVE_QUOTE_ = DecimalMath.divFloor(spareQuote,
94             totalBaseCapital);
95     } else {
96         _TARGET_QUOTE_TOKEN_AMOUNT_ = _QUOTE_BALANCE_;
97     }

```

```

98     if (_BASE_BALANCE_ > _TARGET_BASE_TOKEN_AMOUNT_) {
99         uint256 spareBase = _BASE_BALANCE_.sub(_TARGET_BASE_TOKEN_AMOUNT_);
100         _QUOTE_CAPITAL_RECEIVE_BASE_ = DecimalMath.divFloor(spareBase,
101             totalQuoteCapital);
102     } else {
103         _TARGET_BASE_TOKEN_AMOUNT_ = _BASE_BALANCE_;
104     }
105     _R_STATUS_ = Types.RStatus.ONE;
106 }

```

Listing 3.10: contracts/impl/Settlement.sol

Recommendation Add necessary decentralized mechanisms to reduce or separate overly centralized privileges around `_OWNER_`. In the meantime, develop a long-time plan for eventual community-based governance.

3.7 Possible Integer Overflow in `sqrt()`

- ID: PVE-007
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: DODOMath
- Category: Numeric Errors [14]
- CWE subcategory: CWE-190 [5]

Description

In previous sections, we have discussed two quadratic solutions, i.e., `_SolveQuadraticFunctionForTrade()` and `_SolveQuadraticFunctionForTrade()`, behind the PMM algorithm. Note each of these two quadratic solutions requires the capability to calculate the integer square root of a given number, i.e., the familiar `sqrt()` function. The `sqrt()` function, implemented in `SafeMath`, follows the Babylonian method for calculating the integer square root. Specifically, for a given x , we need to find out the largest integer z such that $z^2 \leq x$.

```

55     function sqrt(uint256 x) internal pure returns (uint256 y) {
56         uint256 z = (x + 1) / 2;
57         y = x;
58         while (z < y) {
59             y = z;
60             z = (x / z + z) / 2;
61         }
62     }

```

Listing 3.11: contracts/lib/SafeMath.sol

We show above current `sqrt()` implementation. The initial value of `z` to the iteration was given as $z = (x + 1)/2$, which results in an integer overflow when $x = \text{uint256}(-1)$. In other words, the overflow essentially sets `z` to zero, leading to a division by zero in the calculation of $z = (x/z + z)/2$ (line 60).

Note that this does not result in an incorrect return value from `sqrt()`, but does cause the function to revert unnecessarily when the above corner case occurs. Meanwhile, it is worth mentioning that if there is a divide by zero, the execution or the contract call will be thrown by executing the `INVALID` opcode, which by design consumes all of the gas in the initiating call. This is different from `REVERT` and has the undesirable result in causing unnecessary monetary loss.

To address this particular corner case, We suggest to change the initial value to $z = x/2 + 1$, making `sqrt()` well defined over its all possible inputs.

Recommendation Revise the above calculation to avoid the unnecessary integer overflow.

3.8 Redundant State Checks

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Pricing
- Category: Security Features [11]
- CWE subcategory: CWE-269 [6]

Description

The PMM algorithm has its continuous price curve with three different operating states: `ROne`, `RAbove`, and `RBelow`. For each state, DODO specifies two trading functions to accommodate users' requests, i.e., `BuyBaseToken` and `SellBaseToken`. In total, there are six trading functions: `_ROneBuyBaseToken()`, `_ROneSellBaseToken()`, `_RAboveBuyBaseToken()`, `_RAboveSellBaseToken()`, `_RBelowBuyBaseToken()`, and `_RBelowSellBaseToken()`.

In `_ROneBuyBaseToken()`, we notice that the `require` check on `amount < targetBaseTokenAmount` (line 51) is redundant. The reason is that the subsequent `sub` operation (line 52) applies the same validity check (in `SafeMath.sol`, line 45). Similarly, the `require` check (line 118) on `amount < baseBalance` is also redundant since the subsequent `sub` operation (line 119) applies the very same validity check.

```

46     function _ROneBuyBaseToken(uint256 amount, uint256 targetBaseTokenAmount)
47         internal
48         view
49         returns (uint256 payQuoteToken)
50     {
51         require(amount < targetBaseTokenAmount, "DODO_BASE_TOKEN_BALANCE_NOT_ENOUGH");

```

```

52     uint256 B2 = targetBaseTokenAmount.sub(amount);
53     payQuoteToken = _RAboveIntegrate(targetBaseTokenAmount, targetBaseTokenAmount,
54                                     B2);
55     return payQuoteToken;

```

Listing 3.12: contracts/impl/Pricing.sol

```

111 // ===== R > 1 cases =====
112
113 function _RAboveBuyBaseToken(
114     uint256 amount,
115     uint256 baseBalance,
116     uint256 targetBaseAmount
117 ) internal view returns (uint256 payQuoteToken) {
118     require(amount < baseBalance, "DODO_BASE_TOKEN_BALANCE_NOT_ENOUGH");
119     uint256 B2 = baseBalance.sub(amount);
120     return _RAboveIntegrate(targetBaseAmount, baseBalance, B2);
121 }

```

Listing 3.13: contracts/impl/Pricing.sol

Recommendation Optionally remove these redundant checks. Note that this is optional as the error message conveys additional semantic information when the intended `revert` occurs.

3.9 Contract Verification in breedDODO()

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: DODOZoo
- Category: Security Features [11]
- CWE subcategory: CWE-269 [6]

Description

The smart contract DODOZoo is in charge of registering all token pairs available for trading in DODO. Note that adding a new pair of `baseToken` and `quoteToken` is a privileged operation only allowed by the contract owner (with the `onlyOwner` modifier).¹ However, the registration of a new token pair does not perform a thorough validity check on the tokens of the given pair. Though an offline check can be performed by the owner, it is still suggested to perform the necessary check codified at the contract to verify the given `baseToken` and `quoteToken` are indeed intended ERC20 token contracts. This is necessary as DODO users recognize and trust both `baseToken` and `quoteToken` before interacting with

¹The removal of a registered token pair is not possible. For that, the DODO protocol only allows the registered pair to be closed or settled from being further tradable.

them. To ensure the two token addresses are not provided accidentally, a non-zero `extcodesize` check could be added.

```

29 // ===== Breed DODO Function =====
30
31 function breedDODO(
32     address supervisor ,
33     address maintainer ,
34     address baseToken ,
35     address quoteToken ,
36     address oracle ,
37     uint256 lpFeeRate ,
38     uint256 mtFeeRate ,
39     uint256 k ,
40     uint256 gasPriceLimit
41 ) public onlyOwner returns (address) {
42     require(!isDODOResistered(baseToken , quoteToken) , "DODO_IS_REGISTERED");
43     require(baseToken != quoteToken , "BASE_IS_SAME_WITH_QUOTE");
44     address newBornDODO = address(new DODO());
45     IDODO(newBornDODO).init(
46         supervisor ,
47         maintainer ,
48         baseToken ,
49         quoteToken ,
50         oracle ,
51         lpFeeRate ,
52         mtFeeRate ,
53         k ,
54         gasPriceLimit
55     );
56     IDODO(newBornDODO).transferOwnership(_OWNER_);
57     _DODO_REGISTER_[baseToken][quoteToken] = newBornDODO;
58     emit DODOBirth(newBornDODO);
59     return newBornDODO;
60 }

```

Listing 3.14: contracts/DODOZoo.sol

Moreover, the event generation in line 58 can be further improved by including both `baseToken` and `quoteToken`, i.e., `emit DODOBirth(newBornDODO, baseToken, quoteToken)`.

Recommendation Validate that the given token pairs (`baseToken` and `quoteToken`) are indeed ERC20 tokens.

3.10 Balance Inconsistency With Deflationary Tokens

- ID: PVE-011
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Trader, LiquidityProvider
- Category: Time and State [12]
- CWE subcategory: CWE-362 [8]

Description

DODO acts as a trustless intermediary between liquidity providers and trading users. The liquidity providers deposit either `baseToken` or `quoteToken` into the DODO pool and in return get the tokenized share `DODOLpToken` of the pool's assets. Later on, the liquidity providers can `withdraw` their own share by returning `DODOLpToken` back to the pool. With assets in the pool, users can submit `trade` orders and the trading price is determined according to the PMM price curve.

For the above three operations, i.e., `deposit`, `withdraw`, and `trade`, DODO provides low-level routines to transfer assets into or out of the pool (see the code snippet below). These asset-transferring routines work as expected with standard ERC20 tokens: namely DODO's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```

38     function _baseTokenTransferIn(address from, uint256 amount) internal {
39         IERC20(_BASE_TOKEN_).safeTransferFrom(from, address(this), amount);
40         _BASE_BALANCE_ = _BASE_BALANCE_.add(amount);
41     }
42
43     function _quoteTokenTransferIn(address from, uint256 amount) internal {
44         IERC20(_QUOTE_TOKEN_).safeTransferFrom(from, address(this), amount);
45         _QUOTE_BALANCE_ = _QUOTE_BALANCE_.add(amount);
46     }
47
48     function _baseTokenTransferOut(address to, uint256 amount) internal {
49         IERC20(_BASE_TOKEN_).safeTransfer(to, amount);
50         _BASE_BALANCE_ = _BASE_BALANCE_.sub(amount);
51     }
52
53     function _quoteTokenTransferOut(address to, uint256 amount) internal {
54         IERC20(_QUOTE_TOKEN_).safeTransfer(to, amount);
55         _QUOTE_BALANCE_ = _QUOTE_BALANCE_.sub(amount);
56     }

```

Listing 3.15: `contracts/impl/Settlement.sol`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer` or

`transferFrom`. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit`, `withdraw`, and `trade`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of DODO and affects protocol-wide operation and maintenance.

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `transfer` or `transferFrom` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer`/`transferFrom` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into DODO for trading. In current implementation, DODO requires the `owner` privilege to permit tradable ERC20 tokens. By doing so, it may eliminate such concern, but it completely depends on privileged accounts. Such dependency not only affects the intended eventual decentralization, but also limits the number or scale of pairs supported for rapid, mass adoption of DODO.

Recommendation Apply necessary mitigation mechanisms to regulate non-compliant or unnecessarily-extended ERC20 tokens.

3.11 Aggregated Transfer of Maintainer Fees

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DODOZoo
- Category: Security Features [11]
- CWE subcategory: CWE-269 [6]

Description

DODO has a protocol-wide fee parameter `_MT_FEE_RATE_` that is designed to offset the protocol development, operation, management and other associated cost. The fee or `maintainer fee` is not applied up-front before trading. Instead, it is applied after calculating the trading result. For example, if a user attempts to `sellBaseToken`, the fee is deducted from `receiveQuote` and the exact amount is calculated as `mtFeeQuote = DecimalMath.mul(receiveQuote, _MT_FEE_RATE_)`. Similarly, if a user attempts to `buyBaseToken`, the fee is deducted from `payQuote` and the exact amount is calculated as `mtFeeBase = DecimalMath.mul(amount, _MT_FEE_RATE_)`.

We notice that the maintainer fee is calculated and collected for each trade as part of the trade processing. The immediate transfer-out of the calculated `maintainer fee` at the time of the trade

would impose an additional gas cost on every trade. To avoid this, accumulated `maintainer fees` can be temporarily saved on the DODO contract and later retrieved by authorized entity. Considering the scenario of having a deflationary token on the trading token pair, each trade may generate a very small number of `maintainer fee`. Apparently, it is economically desirable to first accumulate the `maintainer fees` and, after the fees reach a certain amount, then perform an aggregated transfer-out.

```

84     function buyBaseToken(uint256 amount, uint256 maxPayQuote)
85         external
86         tradeAllowed
87         gasPriceLimit
88         preventReentrant
89         returns (uint256)
90     {
91         // query price
92         (
93             uint256 payQuote ,
94             uint256 lpFeeBase ,
95             uint256 mtFeeBase ,
96             Types.RStatus newRStatus ,
97             uint256 newQuoteTarget ,
98             uint256 newBaseTarget
99         ) = _queryBuyBaseToken(amount);
100         require(payQuote <= maxPayQuote, "BUY_BASE_COST_TOO_MUCH");
101
102         // settle assets
103         _quoteTokenTransferIn(msg.sender, payQuote);
104         _baseTokenTransferOut(msg.sender, amount);
105         _baseTokenTransferOut(_MAINTAINER_, mtFeeBase);
106
107         // update TARGET
108         _TARGET_QUOTE_TOKEN_AMOUNT_ = newQuoteTarget;
109         _TARGET_BASE_TOKEN_AMOUNT_ = newBaseTarget;
110         _R_STATUS_ = newRStatus;
111
112         _donateBaseToken(lpFeeBase);
113         emit BuyBaseToken(msg.sender, amount, payQuote);
114         emit MaintainerFee(true, mtFeeBase);
115
116         return payQuote;
117     }

```

Listing 3.16: `contracts/impl/trader.sol`

Recommendation Avoid transferring the maintainer fee for each trade. Instead, accumulate the maintainer fee within the contract and allow the authorized entity to aggregately withdraw the fee to the right maintainer.

3.12 Misleading Embedded Code Comments

- ID: PVE-013
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: DODOMath, Pricing
- Category: Coding Practices [13]
- CWE subcategory: CWE-1116 [4]

Description

There are a few misleading comments embedded among lines of solidity code, which may introduce unnecessary burdens to understand or maintain the software.

A few example comments can be found in lines 73 and 88 of `lib/DODOMath::_SolveQuadraticFunctionForTrade()`, and line 140 of `impl/Pricing::_RAboveBackToOne()`. We show below the quadratic solution function `_SolveQuadraticFunctionForTrade()`.

```

58     function _SolveQuadraticFunctionForTrade(
59         uint256 Q0,
60         uint256 Q1,
61         uint256 ideltaB ,
62         bool deltaBSig ,
63         uint256 k
64     ) internal pure returns (uint256) {
65         // calculate -b value and sig
66         // -b = (1-k)Q1-kQ0^2/Q1+i*deltaB
67         uint256 kQ02Q1 = DecimalMath.mul(k, Q0).mul(Q0).div(Q1); // kQ0^2/Q1
68         uint256 b = DecimalMath.mul(DecimalMath.ONE.sub(k), Q1); // (1-k)Q1
69         bool minusbSig = true;
70         if (deltaBSig) {
71             b = b.add(ideltaB); // (1-k)Q1+i*deltaB
72         } else {
73             kQ02Q1 = kQ02Q1.add(ideltaB); // -i*(-deltaB)-kQ0^2/Q1
74         }
75         if (b >= kQ02Q1) {
76             b = b.sub(kQ02Q1);
77             minusbSig = true;
78         } else {
79             b = kQ02Q1.sub(b);
80             minusbSig = false;
81         }
82
83         // calculate sqrt
84         uint256 squareRoot = DecimalMath.mul(
85             DecimalMath.ONE.sub(k).mul(4),
86             DecimalMath.mul(k, Q0).mul(Q0)
87         ); // 4(1-k)kQ0^2
88         squareRoot = b.mul(b).add(squareRoot).sqrt(); // sqrt(b*b-4(1-k)kQ0*Q0)
89     }

```

```

90 // final res
91 uint256 denominator = DecimalMath.ONE.sub(k).mul(2); // 2(1-k)
92 if (minusbSig) {
93     return DecimalMath.divFloor(b.add(squareRoot), denominator);
94 } else {
95     return DecimalMath.divFloor(squareRoot.sub(b), denominator);
96 }
97 }

```

Listing 3.17: contracts/lib/DODOMath.sol

The comment in line 73 is supposed to be `// i*deltaB+kQ0^2/Q1` while the comment in line 88 should be `// sqrt(b*b-4*(1-k)*(-kQ0*Q0))`.

Recommendation Adjust the comments accordingly (e.g., from “`sqrt(b*b-4*(1-k)kQ0*Q0)`” to “`sqrt(b*b-4*(1-k)*(-kQ0*Q0))`” in `DODOMath.sol` - line 88).

```

83 // calculate sqrt
84 uint256 squareRoot = DecimalMath.mul(
85     DecimalMath.ONE.sub(k).mul(4),
86     DecimalMath.mul(k, Q0).mul(Q0)
87 ); // 4(1-k)kQ0^2
88 squareRoot = b.mul(b).add(squareRoot).sqrt(); // sqrt(b*b-4*(1-k)*(-kQ0*Q0))

```

Listing 3.18: contracts/lib/DODOMath.sol

3.13 Missing DODO Validation in DODOEthProxy

- ID: PVE-013
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: DODOEthProxy
- Category: Coding Practices [13]
- CWE subcategory: CWE-628 [9]

Description

To accommodate the support of ETH, DODO provides a proxy contract `DODOEthProxy` that conveniently wraps ETH into WETH and unwraps WETH back to ETH. The goal here is to allow for the unified handling of ETH just like other standard ERC20 tokens.

```

75 function getDODO(address baseToken, address quoteToken) external view returns (
76     address) {
77     return _DODO_REGISTER_[baseToken][quoteToken];
78 }

```

Listing 3.19: contracts/DODOZoo.sol

The smart contract `DODOEthProxy` has three main public methods: `sellEthTo()`, `buyEthWith()`, and `depositEth()`. All these three methods query `DODOZoo` for the presence of a DODO contract being requested. Notice that the query function `getDODO()` returns the requested registered entry. If the entry does not exist, it simply returns back `address(0)`. Unfortunately, if a user somehow provides a wrong `quoteTokenAddress`, the above three methods will be interacting with `address(0)`, leading to possible ETH loss from the user. In the following, we use `sellEthTo()` to elaborate the possible unintended consequence.

If we delve into the `sellEthTo()` routine, the variable `DODO` (line 70) contains the queried DODO address from `DODOZoo`. If the aforementioned corner case occurs, it simply contains `address(0)`. As a result, the subsequent operations essentially become `no op`, except it continues to deposit `ethAmount` into `_WETH` (lines 73–74). These deposited ETHs are not credited and therefore become withdrawable by any one through `buyEthWith()` (that does not validate the presence of DODO either).

```

64     function sellEthTo(
65         address quoteTokenAddress,
66         uint256 ethAmount,
67         uint256 minReceiveTokenAmount
68     ) external payable preventReentrant returns (uint256 receiveTokenAmount) {
69         require(msg.value == ethAmount, "ETH_AMOUNT_NOT_MATCH");
70         address DODO = IDODOZoo(_DODO_ZOO_).getDODO(_WETH_, quoteTokenAddress);
71         receiveTokenAmount = IDODO(DODO).querySellBaseToken(ethAmount);
72         require(receiveTokenAmount >= minReceiveTokenAmount, "RECEIVE_NOT_ENOUGH");
73         IWETH(_WETH_).deposit{value: ethAmount}();
74         IWETH(_WETH_).approve(DODO, ethAmount);
75         IDODO(DODO).sellBaseToken(ethAmount, minReceiveTokenAmount);
76         _transferOut(quoteTokenAddress, msg.sender, receiveTokenAmount);
77         emit ProxySellEth(msg.sender, quoteTokenAddress, ethAmount, receiveTokenAmount);
78         return receiveTokenAmount;
79     }

```

Listing 3.20: `contracts/DODOEthProxy.sol`

Recommendation Validate the `DODO` variable and require its existence (not `address(0)`) before continuing the process.

3.14 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version inconsistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.5.12`; instead of `pragma solidity ^0.5.12`;

Moreover, we strongly suggest not to use experimental Solidity features (e.g., `pragma experimental ABIEncoderV2`) or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, 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.



4 | Conclusion

In this audit, we thoroughly analyzed the DODO documentation and implementation. The audited system presents a unique innovation and we are really impressed by the design and implementation. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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.



5 | Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [17, 18, 19, 20, 22].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [23] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected revert.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External Call

- Description: Whether the contract has any external call without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless Send

- Description: Whether the contract is vulnerable to gasless send.
- Result: Not found
- Severity: Medium

5.1.12 Send Instead Of Transfer

- Description: Whether the contract uses send instead of transfer.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to Out-Of-Gas exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low



References

- [1] axic. Enforcing ABI length checks for return data from calls can be breaking. <https://github.com/ethereum/solidity/issues/4116>.
- [2] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. <https://github.com/ethereum/EIPs/issues/738>.
- [3] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. <https://cwe.mitre.org/data/definitions/1099.html>.
- [4] MITRE. CWE-1116: Inaccurate Comments. <https://cwe.mitre.org/data/definitions/1116.html>.
- [5] MITRE. CWE-190: Integer Overflow or Wraparound. <https://cwe.mitre.org/data/definitions/190.html>.
- [6] MITRE. CWE-269: Improper Privilege Management. <https://cwe.mitre.org/data/definitions/269.html>.
- [7] MITRE. CWE-282: Improper Ownership Management. <https://cwe.mitre.org/data/definitions/282.html>.
- [8] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). <https://cwe.mitre.org/data/definitions/362.html>.
- [9] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. <https://cwe.mitre.org/data/definitions/628.html>.

-
- [10] MITRE. CWE-654: Reliance on a Single Factor in a Security Decision. <https://cwe.mitre.org/data/definitions/654.html>.
 - [11] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
 - [12] MITRE. CWE CATEGORY: 7PK - Time and State. <https://cwe.mitre.org/data/definitions/361.html>.
 - [13] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
 - [14] MITRE. CWE CATEGORY: Numeric Errors. <https://cwe.mitre.org/data/definitions/189.html>.
 - [15] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
 - [16] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
 - [17] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). <https://www.peckshield.com/2018/04/22/batchOverflow/>.
 - [18] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). <https://www.peckshield.com/2018/05/18/burnOverflow/>.
 - [19] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). <https://www.peckshield.com/2018/05/10/multiOverflow/>.
 - [20] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). <https://www.peckshield.com/2018/04/25/proxyOverflow/>.
 - [21] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
 - [22] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. <https://www.peckshield.com/2018/04/28/transferFlaw/>.

- [23] Solidity. Warnings of Expressions and Control Structures. <http://solidity.readthedocs.io/en/develop/control-structures.html>.

