SMART CONTRACT AUDIT REPORT

for

DODO LimitOrder

Prepared By: Yiqun Chen

PeckShield

December 30, 2021

Document Properties

Client	DODO
Title	Smart Contract Audit Report
Target	DODO LimitOrder
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 30, 2021	Xiaotao Wu	Final Release
1.0-rc2	December 24, 2021	Xiaotao Wu	Release Candidate #2
1.0-rc1	December 17, 2021	Xiaotao Wu	Release Candidate #1

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 source code of the `DODO LimitOrder` protocol, 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 LimitOrder

`DODO LimitOrder` realizes the limit order and instantaneous token-exchange functions from the business level. For the price limit order business, users can limit the price at the front end. When the price is satisfied, the `DODO` back end service will perform settlement on the chain, enable users to buy corresponding tokens according to the specified price. For instantaneous token-exchange, `DODO` provides market makers with corresponding `API` interfaces. Users are matched with market makers at the front end and can be matched by `DODO`.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of DODO LimitOrder

Item	Description
Name	DODO
Website	https://dodoex.io/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 30, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/DODOEX/dodo-limit-order.git (a94248a)

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

- https://github.com/DODOEX/dodo-limit-order.git (a35d573)

1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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.

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) [8], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `DODO LimitOrder` protocol smart contracts. 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	1	■
Medium	1	■
Low	2	■ ■
Informational	1	■
Total	5	

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 1 high-severity vulnerability, 1 medium-severity vulnerability 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Inconsistent Implementation Between matchingRFQByPlatform() And fillRFQByUser()	Coding Practices	Fixed
PVE-002	Informational	Meaningful Events For Important State Changes	Coding Practices	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-005	High	Improved Logic In DODOLimitOrder::fillRFQByUser()	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 Inconsistent Implementation Between matchingRFQByPlatform() And fillRFQByUser()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `DODOLimitOrder`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In the `DODOLimitOrder` contract, the request for quotation (`RFQ`) can be filled either by a user or by the platform. When a `RFQ` is filled by the platform, a certain amount of fee will be charged. While reviewing the implementation of the `matchingRFQByPlatform()` routine, we notice that there exists certain inconsistency that can be resolved.

To elaborate, we show below its code snippet. It comes to our attention that the `taker` of an order can be changed arbitrarily by the function caller (line 149). Compared with the implementation of function `fillRFQByUser()`, the `order.taker` should be allowed to change only when `order.taker != address(0)`.

```
134     function matchingRFQByPlatform(
135         Order memory order,
136         bytes memory makerSignature,
137         bytes memory takerSignature,
138         uint256 takerFillAmount,
139         uint256 thresholdMakerAmount,
140         uint256 makerTokenFeeAmount,
141         address taker
142     ) public returns(uint256 curTakerFillAmount, uint256 curMakerFillAmount) {
143         uint256 filledTakerAmount = _RFQ_FILLED_TAKER_AMOUNT_[order.maker][order.
            saltOrSlot];
144         require(filledTakerAmount < order.takerAmount, "DLOP: ALREADY_FILLED");
145
```

```
        bytes32 orderHashForMaker = _orderHash(order);
        require(ECDSA.recover(orderHashForMaker, makerSignature) == order.maker, "DLOP:
            INVALID_MAKER_SIGNATURE");

        order.taker = taker;
        order.makerTokenFeeAmount = makerTokenFeeAmount;
        bytes32 orderHashForTaker = _orderHash(order);
        require(ECDSA.recover(orderHashForTaker, takerSignature) == taker, "DLOP:
            INVALID_TAKER_SIGNATURE");

        (curTakerFillAmount, curMakerFillAmount) = _settleRFQ(order,filledTakerAmount,
            takerFillAmount,thresholdMakerAmount,taker);

        emit RFQByPlatformFilled(order.maker, taker, orderHashForMaker,
            curTakerFillAmount, curMakerFillAmount);
    }
```

Listing 3.1: `DODOLimitOrder::matchingRFQByPlatform()`

Recommendation Validate the input parameter value of above mentioned function to ensure `order.taker != address(0)`.

Status This issue has been fixed in the following commit: `a8fa0d6`.

3.2 Meaningful Events For Important State Changes

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `DODOLimitOrder`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `DODOLimitOrder` contract as an example. While examining the events that reflect the `DODOLimitOrder` dynamics, we notice there is a lack of emitting related events to reflect important state changes. Specifically, when the `addWhiteList()/removeWhiteList()/changeFeeReceiver()` functions are being called, there are no corresponding events being emitted to reflect the occurrence of `addWhiteList()/removeWhiteList()/changeFeeReceiver()`.

```
    //============  Ownable ============
    function addWhiteList (address contractAddr) public onlyOwner {
        isWhiteListed[contractAddr] = true;
    }

    function removeWhiteList (address contractAddr) public onlyOwner {
        isWhiteListed[contractAddr] = false;
    }

    function changeFeeReceiver (address newFeeReceiver) public onlyOwner {
        _FEE_RECEIVER_ = newFeeReceiver;
    }
```

Listing 3.2: `DODOLimitOrder::addWhiteList()/removeWhiteList()/changeFeeReceiver()`

Recommendation Properly emit the related events when the above-mentioned functions are being invoked.

Status This issue has been fixed in the following commit: `a8fa0d6`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `DODOLimitOrder/DODOLimitOrderBot`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `DODO LimitOrder` protocol, there is certain privileged account, i.e., `owner`. When examining the related contracts, we notice inherent trust on this privileged account. To elaborate, we show below the related functions.

Firstly, the privileged functions of the `DODOLimitOrder` contract allow for the `owner` to add/remove contract to the white list or change the `_FEE_RECEIVER_`.

```
    //============  Ownable ============
    function addWhiteList (address contractAddr) public onlyOwner {
        isWhiteListed[contractAddr] = true;
    }

    function removeWhiteList (address contractAddr) public onlyOwner {
        isWhiteListed[contractAddr] = false;
    }

```

```
    function changeFeeReceiver (address newFeeReceiver) public onlyOwner {
        _FEE_RECEIVER_ = newFeeReceiver;
    }
```

Listing 3.3: `DODOLimitOrder::addWhiteList()/removeWhiteList()/changeFeeReceiver()`

Note a contract added to the white list is allowed to call back when this contract calls the `fillLimitOrder()` function (lines 98-99).

```
    // ============= LimitOrder ===============
    function fillLimitOrder(
        Order memory order,
        bytes memory signature,
        uint256 takerFillAmount,
        uint256 thresholdTakerAmount,
        bytes memory takerInteraction
    ) public returns(uint256 curTakerFillAmount, uint256 curMakerFillAmount) {
        bytes32 orderHash = _orderHash(order);
        uint256 filledTakerAmount = _FILLED_TAKER_AMOUNT_[orderHash];

        require(filledTakerAmount < order.takerAmount, "DLOP: ALREADY_FILLED");

        if (order.taker != address(0)) {
            require(order.taker == msg.sender, "DLOP:PRIVATE_ORDER");
        }

        require(ECDSA.recover(orderHash, signature) == order.maker, "DLOP:INVALID_SIGNATURE");
        require(order.expiration > block.timestamp, "DLOP: EXPIRE_ORDER");


        uint256 leftTakerAmount = order.takerAmount.sub(filledTakerAmount);
        curTakerFillAmount = takerFillAmount < leftTakerAmount ? takerFillAmount:leftTakerAmount;
        curMakerFillAmount = curTakerFillAmount.mul(order.makerAmount).div(order.takerAmount);

        require(curTakerFillAmount > 0 && curMakerFillAmount > 0, "DLOP:ZERO_FILL_INVALID");
        require(curTakerFillAmount >= thresholdTakerAmount, "DLOP:FILL_AMOUNT_NOT_ENOUGH");

        _FILLED_TAKER_AMOUNT_[orderHash] = filledTakerAmount.add(curTakerFillAmount);

        //Maker => Taker
        IDODOApproveProxy(_DODO_APPROVE_PROXY_).claimTokens(order.makerToken, order.maker, msg.sender, curMakerFillAmount);

        if(takerInteraction.length > 0) {
            takerInteraction.patchUint256(0, curTakerFillAmount);
            takerInteraction.patchUint256(1, curMakerFillAmount);
            require(isWhiteListed[msg.sender], "DLOP: Not Whitelist Contract");
            (bool success, ) = msg.sender.call(takerInteraction);
```

```
            require(success, "DLOP: TAKER_INTERACTIVE_FAILED");
        }

        //Taker => Maker
        IERC20(order.takerToken).safeTransferFrom(msg.sender, order.maker, curTakerFillAmount);

        emit LimitOrderFilled(order.maker, msg.sender, orderHash, curTakerFillAmount, curMakerFillAmount);
    }
```

Listing 3.4: `DODOLimitOrder::fillLimitOrder()`

Secondly,the privileged functions of the `DODOLimitOrderBot` contract allow for the `owner` to add/remove account to the admin list or change the `_TOKEN_RECEIVER_`..

```
    //============  Ownable ============
    function addAdminList (address userAddr) external onlyOwner {
        isAdminListed[userAddr] = true;
        emit addAdmin(userAddr);
    }

    function removeAdminList (address userAddr) external onlyOwner {
        isAdminListed[userAddr] = false;
        emit removeAdmin(userAddr);
    }

    function changeTokenReceiver(address newTokenReceiver) external onlyOwner {
        _TOKEN_RECEIVER_ = newTokenReceiver;
        emit changeReceiver(newTokenReceiver);
    }
```

Listing 3.5: `DODOLimitOrderBot::addAdminList()/removeAdminList()/changeTokenReceiver()`

Note the `fillDODOLimitOrder()` function call only be called by the accounts that added to the admin list (lines 51).

```
46      function fillDODOLimitOrder(
47          bytes memory callExternalData, //call DODOLimitOrder
48          address takerToken,
49          uint256 minTakerTokenAmount
50       ) external {
51          require(isAdminListed[msg.sender], "ACCESS_DENIED");
52          uint256 originTakerBalance = IERC20(takerToken).balanceOf(address(this));
53
54          (bool success, ) = _DODO_LIMIT_ORDER_.call(callExternalData);
55          require(success, "EXEC_DODO_LIMIT_ORDER_ERROR");
56
57          uint256 takerBalance = IERC20(takerToken).balanceOf(address(this));
58          uint256 leftTakerAmount = takerBalance.sub(originTakerBalance);
59
60          require(leftTakerAmount >= minTakerTokenAmount, "TAKER_AMOUNT_NOT_ENOUGH");
61
```

```
        IERC20(takerToken).transfer(_TOKEN_RECEIVER_, leftTakerAmount);

        //TODO:
        emit Fill();
    }
```

Listing 3.6: `DODOLimitOrderBot::fillDODOLimitOrder()`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the `owner` may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to `owner` explicit to `DODO LimitOrder` users.

Status This issue has been confirmed.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DODOLimitOrderBot`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1109 [1]

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 `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
```

```
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.7: `ZRX.sol`

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, 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 `transferFrom()` as well, i.e., `safeTransferFrom()`

In the following, we show the `fillDODOLimitOrder()` routines in the `DODOLimitOrderBot` contract. If the `ZRX` token is supported as the underlying `IERC20(takerToken)`, the unsafe version of `IERC20(takerToken).transfer(_TOKEN_RECEIVER_, leftTakerAmount)` (line 62) may return false in the `ZRX` token contract's `transfer()` implementation (but the `ERC20` interface expects a revert)! Thus, the contract has vulnerabilities against fake `transfer` attacks.

```
46      function fillDODOLimitOrder(
47          bytes memory callExternalData, //call DODOLimitOrder
48          address takerToken,
49          uint256 minTakerTokenAmount
50       ) external {
51          require(isAdminListed[msg.sender], "ACCESS_DENIED");
52          uint256 originTakerBalance = IERC20(takerToken).balanceOf(address(this));
53
54          (bool success, ) = _DODO_LIMIT_ORDER_.call(callExternalData);
55          require(success, "EXEC_DODO_LIMIT_ORDER_ERROR");
56
57          uint256 takerBalance = IERC20(takerToken).balanceOf(address(this));
58          uint256 leftTakerAmount = takerBalance.sub(originTakerBalance);
59
60          require(leftTakerAmount >= minTakerTokenAmount, "TAKER_AMOUNT_NOT_ENOUGH");
61
62          IERC20(takerToken).transfer(_TOKEN_RECEIVER_, leftTakerAmount);
63
```

```
        //TODO:
        emit Fill();
    }
```

Listing 3.8: `DODOLimitOrderBot::fillDODOLimitOrder()`

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

Status This issue has been fixed in the following commit: `a8fa0d6`.

3.5 Improved Logic In DODOLimitOrder::fillRFQByUser()

- ID: PVE-005
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `DODOLimitOrder`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `DODOLimitOrder` contract provides an public `fillRFQByUser()` function for user to fill a `RFQ` order. While examining the routine, we notice the current implementation logic is flawed.

To elaborate, we show below its code snippet. It comes to our attention that the `taker` of a `RFQ` order can be specified either by the order `maker` or by the caller of the `fillRFQByUser()` function `if (order.taker == address(0))`. Thus there is a risk that a `RFQ` order can be closed without the `taker`'s knowledge if this taker has approved his/her `takerToken` to the `DODOApprove` contract (line 201).

```
110     //================ RFQ ================
111     function fillRFQByUser(
112         Order memory order,
113         bytes memory signature,
114         uint256 takerFillAmount,
115         uint256 thresholdMakerAmount,
116         address taker
117     ) public returns(uint256 curTakerFillAmount, uint256 curMakerFillAmount) {
118         uint256 filledTakerAmount = _RFQ_FILLED_TAKER_AMOUNT_[order.maker][order.
                saltOrSlot];
119 
120         require(filledTakerAmount < order.takerAmount, "DLOP: ALREADY_FILLED");
121 
122         if (order.taker != address(0)) {
123             require(order.taker == taker, "DLOP:TAKER_INVALID");
124         }
125 
126         bytes32 orderHash = _orderHash(order);
```

```
        require(ECDSA.recover(orderHash, signature) == order.maker, "DLOP: INVALID_SIGNATURE");

        (curTakerFillAmount, curMakerFillAmount) = _settleRFQ(order,filledTakerAmount, takerFillAmount,thresholdMakerAmount,taker);

        emit RFQByUserFilled(order.maker, taker, orderHash, curTakerFillAmount, curMakerFillAmount);
    }
```

Listing 3.9: `DODOLimitOrder::fillRFQByUser()`

```
    //============  internal ============
    function _settleRFQ(
        Order memory order,
        uint256 filledTakerAmount,
        uint256 takerFillAmount,
        uint256 thresholdMakerAmount,
        address taker
    ) internal returns(uint256,uint256) {
        require(order.expiration > block.timestamp, "DLOP: EXPIRE_ORDER");

        uint256 leftTakerAmount = order.takerAmount.sub(filledTakerAmount);
        if(takerFillAmount > leftTakerAmount) {
            return (0,0);
        }

        uint256 curTakerFillAmount = takerFillAmount;
        uint256 curMakerFillAmount = curTakerFillAmount.mul(order.makerAmount).div(order.takerAmount);

        require(curTakerFillAmount > 0 && curMakerFillAmount > 0, "DLOP: ZERO_FILL_INVALID");
        require(curMakerFillAmount.sub(order.makerTokenFeeAmount) >= thresholdMakerAmount, "DLOP: FILL_AMOUNT_NOT_ENOUGH");

        _RFQ_FILLED_TAKER_AMOUNT_[order.maker][order.saltOrSlot] = filledTakerAmount.add(curTakerFillAmount);

        if(order.makerTokenFeeAmount > 0) {
            IDODOApproveProxy(_DODO_APPROVE_PROXY_).claimTokens(order.makerToken, order.maker, _FEE_RECEIVER_, order.makerTokenFeeAmount);
        }
        //Maker => Taker
        IDODOApproveProxy(_DODO_APPROVE_PROXY_).claimTokens(order.makerToken, order.maker, taker, curMakerFillAmount.sub(order.makerTokenFeeAmount));
        //Taker => Maker
        IDODOApproveProxy(_DODO_APPROVE_PROXY_).claimTokens(order.takerToken, taker, order.maker, curTakerFillAmount);

        return (curTakerFillAmount, curMakerFillAmount);
```

```
    }
```

Listing 3.10: `DODOLimitOrder::_settleRFQ()`

Recommendation Correct the above implementation to ensure the taker indeed approves the order.

Status This issue has been fixed in the following commit: `a35d573`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `DODO LimitOrder` protocol. `DODO LimitOrder` realizes the limit order and instantaneous token-exchange functions from the business level. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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

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