# SMART CONTRACT AUDIT REPORT

for

# DODO NFTPOOL

Prepared By: Yiqun Chen

PeckShield

September 24, 2021

## Document Properties

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

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | September 24, 2021 | Xuxian Jiang | Final Release |
| 1.0-rc | September 24, 2021 | Xuxian Jiang | Release Candidate |

## Contact

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


| | |
|---|---|
| Name | Yiqun Chen |
| Phone | +86 183 5897 7782 |
| Email | contact@peckshield.com |

# Contents

# 1 | Introduction

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

DODO is an innovative, next-generation on-chain liquidity provision solution. It recognizes main drawbacks of current `AMM` algorithms (especially in provisioning unstable portfolios and having relatively low funding utilization rates), and accordingly proposes a `Proactive Market Maker (PMM)` algorithm that imitates human market makers to bring sufficient on-chain liquidity. The audited `DODO NFTPool` is a price discovery and liquidity protocol for non-standard assets. Powered by the `PMM` algorithm, it represents a brand new pricing and liquidity solution for the `non-fungible token (NFT)` marketplace.

The basic information of DODO NFTPool is as follows:

Table 1.1: Basic Information of DODO NFTPool

| Item | Description |
|---|---|
| Name | DODO |
| Website | https://app.dodoex.io/ |
| Type | Ethereum Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | September 24, 2021 |

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the repository has a number of subdirectories and this audit mainly focuses on the `NFTPool` subdirectory (under the `feature/nftPool` branch).

- https://github.com/DODOEX/contractV2.git (ee68b8e)

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

- https://github.com/DODOEX/contractV2.git (14a95a6)

## 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.

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.

## 1.4 Disclaimer

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

Table 1.3: The Full List of Check Items

| Category | Check Item |
|---|---|
| **Basic Coding Bugs** | Constructor Mismatch |
| | Ownership Takeover |
| | Redundant Fallback Function |
| | Overflows & Underflows |
| | Reentrancy |
| | Money-Giving Bug |
| | Blackhole |
| | Unauthorized Self-Destruct |
| | Revert DoS |
| | Unchecked External Call |
| | Gasless Send |
| | Send Instead Of Transfer |
| | Costly Loop |
| | (Unsafe) Use Of Untrusted Libraries |
| | (Unsafe) Use Of Predictable Variables |
| | Transaction Ordering Dependence |
| | Deprecated Uses |
| **Semantic Consistency Checks** | Semantic Consistency Checks |
| **Advanced DeFi Scrutiny** | Business Logics Review |
| | Functionality Checks |
| | Authentication Management |
| | Access Control & Authorization |
| | Oracle Security |
| | Digital Asset Escrow |
| | Kill-Switch Mechanism |
| | Operation Trails & Event Generation |
| | ERC20 Idiosyncrasies Handling |
| | Frontend-Contract Integration |
| | Deployment Consistency |
| | Holistic Risk Management |
| **Additional Recommendations** | Avoiding Use of Variadic Byte Array |
| | Using Fixed Compiler Version |
| | Making Visibility Level Explicit |
| | Making Type Inference Explicit |
| | Adhering To Function Declaration Strictly |
| | Following Other Best Practices |

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

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

# 2 | Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the `DODO NFTPool` design and implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

| Severity | # of Findings | |
|---|---|---|
| Critical | 0 | |
| High | 0 | |
| Medium | 1 | ■ |
| Low | 2 | ■ ■ |
| Informational | 1 | ■ |
| Total | 4 | |

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 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 | Medium | Lack Of Proper toggleFlag Assignment | Business Logic | Fixed |
| PVE-002 | Low | Improved Sanity Checks For System/-Function Parameters | Coding Practices | Fixed |
| PVE-003 | Informational | Avoidance Of Repeated Storage Reads And Writes | Coding Practices | Confirmed |
| PVE-004 | Low | Trust on Admin Keys | Security Features | Mitigated |

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 Lack Of Proper toggleFlag Assignment

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: BaseFilterV1
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

### Description

The DODO NFTPool support has a base BaseFilterV1 contract that is inherited by others to share a number of base functionalities. Among these functionalities, there are a few pricing-related parameters, e.g., _GS_START_IN_, _CR_IN_, and _NFT_IN_TOGGLE_, with their specific setter functions. While examining these setters, we notice the current implementation does not properly assign their toggle settings.

To elaborate, we show below an example changeNFTInPrice() function. This function takes three arguments newGsStart, newCr, and toggleFlag. However, it blindly sets the current _NFT_IN_TOGGLE_ to be true (line 227) regardless of the third toggleFlag argument. Note two other _changeNFTRandomOutPrice () and _changeNFTTargetOutPrice() functions share the same issue.

```
211     function changeNFTInPrice(
212         uint256 newGsStart,
213         uint256 newCr,
214         bool toggleFlag
215     ) external onlySuperOwner {
216         _changeNFTInPrice(newGsStart, newCr, toggleFlag);
217     }
218
219     function _changeNFTInPrice(
220         uint256 newGsStart,
221         uint256 newCr,
222         bool toggleFlag
223     ) internal {
```

```
        require(newCr != 0, "CR_INVALID");
        _GS_START_IN_ = newGsStart;
        _CR_IN_ = newCr;
        _NFT_IN_TOGGLE_ = true;

        emit ChangeNFTInPrice(newGsStart, newCr, toggleFlag);
    }
```

Listing 3.1: BaseFilterV1::changeNFTInPrice()

**Recommendation** Revise the above three functions to properly assign the intended toggleFlag.

**Status** This issue has been fixed in the following commit: 14a95a6.

## 3.2 Improved Sanity Checks For System/Function Parameters

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

### Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The DODO NFTPool protocol is no exception. Specifically, if we examine the FilterAdmin and Controller contracts, they have defined a number of protocol-wide risk parameters, e.g., _FEE_RATE_, nftInFeeRate and nftOutFeeRate. In the following, we show an example routine that allows for their changes.

```
    function setFilterAdminFeeRateInfo(
        address filterAdminAddr,
        uint256 nftInFeeRate,
        uint256 nftOutFeeRate,
        bool isOpen
    ) external onlyOwner {
        FilterAdminFeeRateInfo memory feeRateInfo = FilterAdminFeeRateInfo({
            nftInFeeRate: nftInFeeRate,
            nftOutFeeRate: nftOutFeeRate,
            isOpen: isOpen
        });
        filterAdminFeeRates[filterAdminAddr] = feeRateInfo;

        emit SetFilterAdminFeeRateInfo(filterAdminAddr, nftInFeeRate, nftOutFeeRate,
            isOpen);
    }

```

```
function setGlobalParam(uint256 nftInFeeRate, uint256 nftOutFeeRate) external
    onlyOwner {
    _GLOBAL_NFT_IN_FEE_RATE_ = nftInFeeRate;
    _GLOBAL_NFT_OUT_FEE_RATE_ = nftOutFeeRate;

    emit SetGlobalParam(nftInFeeRate, nftOutFeeRate);
}
```

Listing 3.2: An example`setter` in `Controller`

Our result shows the update logic on the above parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large `_FEE_RATE_` parameter will revert every single mint operation.

In addition, the `FilterERC1155V1::ERC1155TargetOut()` function takes a number of arguments and the function can be improved by validating the first two arguments share the same length.

**Recommendation** Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

**Status** This issue has been fixed in the following commit: `14a95a6`.

## 3.3 Avoidance Of Repeated Storage Reads And Writes

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `FilterERC721V1`, `FilterERC1155V1`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1099 [1]

### Description

It is well-known that the storage in blockchain is expensive and the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. In particular, we use the `FilterERC1155V1::ERC1155RandomOut()` as an example. This routine is designed to randomly select a set of `NFT tokenIds` for sale.

```
function ERC1155RandomOut(uint256 amount, address to)
    external
    preventReentrant
    returns (uint256 paid)
{
```

```
107         (uint256 rawPay , ) = queryNFTRandomOut(amount);
108         paid = IFilterAdmin(_OWNER_).burnFragFrom(to, rawPay);
109         for (uint256 i = 0; i < amount; i++) {
110             uint256 randomNum = _getRandomNum() % _TOTAL_NFT_AMOUNT_;
111             uint256 sum;
112             for (uint256 j = 0; j < _NFT_IDS_.length; j++) {
113                 uint256 tokenId = _NFT_IDS_[j];
114                 sum += _NFT_RESERVE_[tokenId];
115                 if (sum >= randomNum) {
116                     _transferOutERC1155(to, tokenId, 1);
117                     emit RandomOut(tokenId, 1);
118                     break;
119                 }
120             }
121         }

123         emit RandomOutOrder(to, paid);
124     }
```

Listing 3.3: `FilterERC1155V1::ERC1155RandomOut()`

We notice the calculation of the internal `for`-loop (line $109 - 121$) involves repeated storage reads and writes on the `_TOTAL_NFT_AMOUNT_` state. And the number of reads and writes depends on the given input argument of `amount`, which may incur unnecessarily high gas cost for the execution. With that, it will be helpful to gas the storage state in a local variable and avoid the repeated reads/writes on the same storage location.

Note another routine `ERC721RandomOut()` shares the same issue.

**Recommendation** Revise the above routine to avoid repeated reads/writes on the same storage state.

**Status**

## 3.4 Trust Issue of Admin Keys

- ID: PVE-0604
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

### Description

In the `DODO NFTPool` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and filter administration). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and a representative privileged access in current contracts.

```
    function emergencyWithdraw(
        address[] memory nftContract,
        uint256[] memory tokenIds,
        address to
    ) external onlySuperOwner {
        require(nftContract.length == tokenIds.length, "PARAM_INVALID");
        address controller = IFilterAdmin(_OWNER_)._CONTROLLER_();
        require(
            IController(controller).isEmergencyWithdrawOpen(address(this)),
            "EMERGENCY_WITHDRAW_NOT_OPEN"
        );

        for (uint256 i = 0; i < nftContract.length; i++) {
            uint256 tokenId = tokenIds[i];
            if (_NFT_RESERVE_[tokenId] > 0 && nftContract[i] == _NFT_COLLECTION_) {
                uint256 index = getNFTIndexById(tokenId);
                if(index != _NFT_IDS_.length - 1) {
                    uint256 lastTokenId = _NFT_IDS_[_NFT_IDS_.length - 1];
                    _NFT_IDS_[index] = lastTokenId;
                    _TOKENID_IDX_[lastTokenId] = index + 1;
                }
                _NFT_IDS_.pop();
                _NFT_RESERVE_[tokenId] = 0;
                _TOKENID_IDX_[tokenId] = 0;
            }
            IERC721(nftContract[i]).safeTransferFrom(address(this), to, tokenIds[i]);
            emit EmergencyWithdraw(nftContract[i],tokenIds[i],to);
        }
        _TOTAL_NFT_AMOUNT_ = _NFT_IDS_.length;
    }
```

Listing 3.4: `FilterERC721V1::emergencyWithdraw()` Contract

Apparently, if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

**Recommendation** Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks.

Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

**Status** This issue has been confirmed with the team. The team clarifies that these privileged functions are currently managed with a trusted `multi-sig` account.

# 4 Conclusion

In this audit, we have analyzed the documentation and implementation of `DODO NFTPool`. The audited system presents a unique pricing and liquidity solution for the `non-fungible token (NFT)` marketplace. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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

# References


[1] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

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

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