SMART CONTRACT AUDIT REPORT

for

DODOV2

Prepared By: Shuxiao Wang

Hangzhou, China

December 17, 2020

Document Properties

Client	DODO
Title	Smart Contract Audit Report
Target	DODOv2
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xudong Shao, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 17, 2020	Xuxian Jiang	Final Release
1.0-rc	December 15, 2020	Xuxian Jiang	Release Candidate
0.3	December 10, 2020	Xuxian Jiang	Additional Findings #2
0.2	December 7, 2020	Xuxian Jiang	Additional Findings #1
0.1	December 3, 2020	Xuxian Jiang	Initial Draft

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the **DODOv2** design document and related smart contract source code, 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 DODOv2

DODO is an innovative, next-generation on-chain liquidity provision solution. It recognizes main drawbacks of current AMM algorithms (especially in provisioning unstable portfolios and having relatively low funding utilization rates), and accordingly proposes an algorithm that imitates human market makers to bring sufficient on-chain liquidity. 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). DODOv2 improves the first version by further supporting private pools and vending machines and continues to advance the DEX frontline by presenting a rare innovation in the rapidly-evolving DeFi ecosystem.

The basic information of DODOv2 is as follows:

Table 1.1: Basic Information of DODOv2

Item	Description
Issuer	DODO
Website	https://app.dododex.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 17, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. As mentioned earlier, DODOv2 assumes a trusted oracle with timely market price feeds and the oracle itself is not part of this audit.

- https://github.com/DODOEX/contractV2.git (6ba6984)

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 (610baa6)

1.2 About PeckShield

PeckShield Inc. [14] 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 [13]:

- 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) [12], 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 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 DODOv2 Protocol 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	1	■
High	0	
Medium	3	■ ■ ■
Low	3	■ ■ ■
Informational	2	■ ■
Total	9	

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 critical-severity vulnerability, 3 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Consistency Between DODOPrivatePool and DODOVendingMachine	Coding Practices	Fixed
PVE-002	Informational	Suggested immutable Usages For Gas Efficiency	Coding Practices	Fixed
PVE-003	Medium	Possible Costly DLPs From Improper Liquidity Initialization	Time and State	Fixed
PVE-004	Low	Improved Corner Case Handling in _setRState()	Business Logic	Fixed
PVE-005	Low	Improved Sanity Checks For System/Function Parameters	Coding Practices	Fixed
PVE-006	Medium	Trust Issue of Admin Keys Behind DODOApprove	Security Features	Mitigated
PVE-007	Low	ERC20-Compliance Issue in DVMStorage	Business Logic	Partially Fixed
PVE-008	Medium	Trade Permission Bypass With Flashloan	Security Features	Fixed
PVE-009	Critical	Confused Deputy For Fund-Stealing	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 Consistency Between DODOPrivatePool and DODOVendingMachine

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: DODOPrivatePool/DODOVendingMachine
- Category: Coding Practices [10]
- CWE subcategory: CWE-1099 [1]

Description

DODOv2 supports two types of liquidity pools – DODOPrivatePool and DODOVendingMachine. As the names indicate, the first type is a private pool owned by a single entity and the second type is shared by multiple liquidity providers. While they apply the same PMM-based price curve, they have different ways to configure pool-specific risk parameters.

A common functionality among these pools is to _sync() the reserves of baseToken and quoteToken assets according to current balances. For illustration, we show below the respective _sync() routine in these two pools.

```
    function _sync() internal {
        uint256 baseBalance = _BASE_TOKEN_.balanceOf(address(this));
        uint256 quoteBalance = _QUOTE_TOKEN_.balanceOf(address(this));
        if (baseBalance != _BASE_RESERVE_) {
            _BASE_RESERVE_ = baseBalance;
        }
        if (quoteBalance != _QUOTE_RESERVE_) {
            _QUOTE_RESERVE_ = quoteBalance;
        }
    }
```

Listing 3.1: DVMVault::_sync() in DODOVendingMachine

```
    function _sync() internal {
        _BASE_RESERVE_ = _BASE_TOKEN_.balanceOf(address(this));
        _QUOTE_RESERVE_ = _QUOTE_TOKEN_.balanceOf(address(this));
    }
```

Listing 3.2: DPPTrader::_sync() in DODOPrivatePool

We notice that both implementations of `_sync` are different, even though share the same functionality. The `DODOPrivatePool` version is primitive in not taking advantage of gas optimization adopted in the `DODOVendingMachine` version. For consistency as well as future maintenance, it is helpful to share the same implementation.

Recommendation Be consistent in both `DODOPrivatePool` and `DODOVendingMachine` when synchronizing the pool balances.

Status The issue has been fixed in this commit: `7bc6f3e`.

3.2 Suggested immutable Usages For Gas Efficiency

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `DPPFactory`, `DVMFactory`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1099 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that 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. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show the key state variables defined in `DVMFactory` and `DPPFactory`. If there is no need to dynamically update these key state variables, they can be declared as `immutable` for gas efficiency.

```
contract DVMFactory is Ownable {
    // ============ Templates ============

    address public _CLONE_FACTORY_;
    address public _DVM_TEMPLATE_;
    address public _DVM_ADMIN_TEMPLATE_;
    address public _FEE_RATE_MODEL_TEMPLATE_;
    address public _PERMISSION_MANAGER_TEMPLATE_;
    address public _DEFAULT_GAS_PRICE_SOURCE_;
    ...
}
```

Listing 3.3: DVMFactory.sol

```
contract DPPFactory is Ownable {
    // ============ Templates ============

    address public _CLONE_FACTORY_;
    address public _DPP_TEMPLATE_;
    address public _DPP_ADMIN_TEMPLATE_;
    address public _FEE_RATE_MODEL_TEMPLATE_;
    address public _PERMISSION_MANAGER_TEMPLATE_;
    address public _DEFAULT_GAS_PRICE_SOURCE_;
    address public _VALUE_SOURCE_;
    address public _DODO_SMART_APPROVE_;
    ...
}
```

Listing 3.4: DPPFactory.sol

Note that both `DODOPrivatePool` and `DODOVendingMachine` take a proxy-based approach that may limit the advantages of `immutable` states. For that, we can take a so-called `immutable forwarding` pattern, which basically passes the `immutable` states as part of function arguments to avoid storage reads. We realize the current proxy is based on the minimum implementation of transparent proxy (`EIP-1167`), the proposed `immutable forwarding` pattern may require revamping the proxy implementation, which may not be suggested unless the gas consumption is a huge concern.

Recommendation Revisit the state variable definition and make extensive use of `immutable` states.

Status The issue has been fixed in this commit: `fc39f70`.

3.3 Possible Costly DLPs From Improper Liquidity Initialization

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `DVMFunding`
- Category: Time and State [9]
- CWE subcategory: CWE-362 [5]

Description

As mentioned in Section 3.1, DODOv2 supports two types of liquidity pools – `DODOPrivatePool` and `DODOVendingMachine`. The `DODOVendingMachine` pool is shared by multiple liquidity providers. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool token, i.e., `DLP`, extremely expensive and bring hurdles (or even causes loss) for later liquidity providers.

To elaborate, we show below the `buyShares()` routine. This routine is used for liquidity providers to deposit supported assets and get respective `DLP` pool tokens in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    // buy shares [round down]
    function buyShares(address to)
        external
        preventReentrant
        returns (
            uint256 shares,
            uint256 baseInput,
            uint256 quoteInput
        )
    {
        uint256 baseBalance = _BASE_TOKEN_.balanceOf(address(this));
        uint256 quoteBalance = _QUOTE_TOKEN_.balanceOf(address(this));
        uint256 baseReserve = _BASE_RESERVE_;
        uint256 quoteReserve = _QUOTE_RESERVE_;

        baseInput = baseBalance.sub(baseReserve);
        quoteInput = quoteBalance.sub(quoteReserve);
        require(baseInput > 0, "NO_BASE_INPUT");

        // case 1. initial supply
        // w/ consideration of baseReserve == 0 && quoteReserve == 0
        // Note: it is not possible to have balance==0 && totalsupply!=0
        // but it is possible to havereserve>0 && totalSupply==0
        if (totalSupply == 0) {
            shares = baseBalance; //
        } else if (baseReserve > 0 && quoteReserve == 0) {
            // case 2. supply when quote reserve is 0
            shares = baseInput.mul(totalSupply).div(baseReserve);
```

```
57         } else if (baseReserve > 0 && quoteReserve > 0) {
58             // case 3. normal case
59             uint256 baseInputRatio = DecimalMath.divFloor(baseInput, baseReserve);
60             uint256 quoteInputRatio = DecimalMath.divFloor(quoteInput, quoteReserve);
61             uint256 mintRatio = quoteInputRatio < baseInputRatio ? quoteInputRatio :
                   baseInputRatio;
62             shares = DecimalMath.mulFloor(totalSupply, mintRatio);
63         }
64         _mint(to, shares);
65         _sync();
66         emit BuyShares(to, shares, _SHARES_[to]);
67     }
```

Listing 3.5: DVMFunding::buyShares()

Specifically, when the pool is being initialized (line 52), the share value directly takes the value of `baseBalance` (line 53), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shares = baseBalance = 1WEI`. With that, the actor can further deposit a huge amount of both `baseToken` and `quoteToken` assets and next invoke the `_sync()` routine with the goal of making the `DLP` pool token extremely expensive. Note the `_sync()` routine can be invoked by simply calling `sellShares()` routine with 0 shares.

An extremely expensive `DLP` pool token can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of `buyShares()` to defensively calculate the share amount when the pool is being initialized.

Status This issue has been fixed in this commit: `6bdf689`.

3.4 Improved Corner Case Handling in _setRState()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DPPVault`
- Category: Business Logic [11]
- CWE subcategory: CWE-837 [7]

Description

According to the DODOv2's PMM algorithm, its unique price curve is continuous but with two distinct segments and three different operating states: `ROne`, `RAbove`, and `RBelow`. The first state `ROne` 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 `ROne`; 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 `ROne`.

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 `DODOMath`). In particular, `DODOMath` has three operations: one specific integration and two other quadratic solutions. The integration computation, i.e., `_GeneralIntegrate()`, is used in `ROne` and `RAbove` to calculate the expected exchange of `quoteToken` for the trading `baseToken` amount. The quadratic solution `_SolveQuadraticFunctionForTrade()` is used in `ROne` 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 `ROne`.

In the following, we show the `_setRState()` routine that is used in `DODOPrivatePool` to configure or reset current operating states.

```
    function _setRState() internal {
        if (_BASE_RESERVE_ == _BASE_TARGET_ && _QUOTE_RESERVE_ == _QUOTE_TARGET_) {
            _RState_ = PMMPricing.RState.ONE;
        } else if (_BASE_RESERVE_ > _BASE_TARGET_) {
            _RState_ = PMMPricing.RState.BELOW_ONE;
        } else if (_QUOTE_RESERVE_ > _QUOTE_TARGET_) {
            _RState_ = PMMPricing.RState.ABOVE_ONE;
        } else {
            require(false, "R_STATE_WRONG");
        }
    }
```

Listing 3.6: DPPVault::_setRState()

This routine updates the pool state based on internal records of `baseToken` and `quoteToken` assets as well as current balances. However, it fails to be more specific in addressing two possible cases: `_BASE_RESERVE_ > _BASE_TARGET_ && _QUOTE_RESERVE_ < _QUOTE_TARGET_` and `_BASE_RESERVE_ < _BASE_TARGET_ && _QUOTE_RESERVE_ > _QUOTE_TARGET_`. In other words, the above routine is better revised as follows:

```
function _setRState() internal {
    if (_BASE_RESERVE_ == _BASE_TARGET_ && _QUOTE_RESERVE_ == _QUOTE_TARGET_) {
        _RState_ = PMMPricing.RState.ONE;
    } else if (_BASE_RESERVE_ > _BASE_TARGET_ && _QUOTE_RESERVE_ < _QUOTE_TARGET_) {
        _RState_ = PMMPricing.RState.BELOW_ONE;
    } else if (_BASE_RESERVE_ < _BASE_TARGET_ && _QUOTE_RESERVE_ > _QUOTE_TARGET_) {
        _RState_ = PMMPricing.RState.ABOVE_ONE;
    } else {
        require(false, "R_STATE_WRONG");
    }
}
```

Listing 3.7: Revised DPPVault::_setRState()

Recommendation Improve the `_setRState()` routine to be explicit in thoroughly addressing possible cases.

Status The issue has been fixed in this commit: `7bc6f3e`.

3.5 Improved Sanity Checks For System/Function Parameters

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The DODOv2 protocol is no exception. Specifically, if we examine the `DPPStorage` contract, it has defined a number of system-wide risk parameters: `_LP_FEE_RATE_MODEL_`, `_MT_FEE_RATE_MODEL_`, `_K_`, and `_I_`.

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these 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 `_LP_FEE_RATE_MODEL_` and `_MT_FEE_RATE_MODEL_` may revert every trade transaction or bring high trading fee.

In addition, a number of functions can benefit from more rigorous validation on their arguments. For example, the `dodoSwapV2ETHToToken()` (see the code below) can be improved by requiring both `dodoPairs` and `directions` have the same length. The same issue is also applicable in `DODV2Proxy01::dodoSwapV2TokenToETH()`, `DODV2Proxy01::dodoSwapV2TokenToToken()`, and `DODV1Proxy01::dodoSwapV1()`.

```
    function dodoSwapV2ETHToToken(
        address payable assetTo,
        address toToken,
        uint256 minReturnAmount,
        address[] memory dodoPairs,
        uint8[] memory directions,
        uint256 deadLine
    )
        external
        virtual
        override
        payable
        judgeExpired(deadLine)
        returns (uint256 returnAmount)
    {
        uint256 originToTokenBalance = IERC20(toToken).balanceOf(msg.sender);

        IWETH(_WETH_).deposit{value: msg.value}();
        IWETH(_WETH_).transfer(dodoPairs[0], msg.value);
        ...
    }
```

Listing 3.8: DODV2Proxy01::dodoSwapV2ETHToToken()

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been fixed in this commit: `95665db`.

3.6 Trust Issue of Admin Keys Behind DODOApprove

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: DODOApprove
- Category: Security Features [8]
- CWE subcategory: CWE-287 [4]

Description

In DODOv2, there is a privileged contract, i.e., DODOApprove, that plays a critical role in receiving allowance from trading users. This contract is designed to greatly facilitate the asset transfers for various swap operations.

In the following, we show the contract implementation. This contract has three functions, i.e., setDODOProxy(), getDODOProxy(), and claimTokens(). The first two are used to set up and query current _DODO_PROXY_ while the last one is used to facilitate asset transfers.

```
contract DODOApprove is Ownable {
    using SafeERC20 for IERC20;
    address public _DODO_PROXY_;

    // ============ Events ============

    event SetDODOProxy(address indexed oldProxy, address indexed newProxy);

    function setDODOProxy(address newDodoProxy) external onlyOwner {
        emit SetDODOProxy(_DODO_PROXY_, newDodoProxy);
        _DODO_PROXY_ = newDodoProxy;
    }

    function getDODOProxy() public view returns (address) {
        return _DODO_PROXY_;
    }

    function claimTokens(
        address token,
        address who,
        address dest,
        uint256 amount
    ) external {
        require(msg.sender == _DODO_PROXY_, "DODOApprove:Access restricted");
        if (amount > 0) {
            IERC20(token).safeTransferFrom(who, dest, amount);
        }
    }
}
```

Listing 3.9: DODOApprove.sol

With the third function, i.e., `claimTokens()`, the current `_DODO_PROXY_` is capable of taking assets from current trading users up to permitted allowances. Fortunately, the first function, i.e., `_setDODOProxy()`, is guarded with the `onlyOwner` modifier, which brings the necessary trust on the `Owner`.

As a mitigation, instead of having a single EOA account as the `Owner`, an alternative is to make use of a multi-sig wallet. To further eliminate the administration key concern, it may be required to transfer the role to a community-governed DAO. In the meantime, a timelock-based mechanism might also be applicable for mitigation.

Recommendation Promptly transfer the `Owner` privilege to the intended `DAO`-like governance contract. And 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 and partially mitigated with additional timelock-based schemes to regulate the owner privileges. The related fixup can be found in this commit: `6bdf689`.

3.7 ERC20-Compliance Issue in DVMStorage

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DVMStorage`
- Category: Business Logic [11]
- CWE subcategory: CWE-754 [6]

Description

In DODOv2, the `DODOVendingMachine` pool implements an ERC20-compliant pool token that represents the ownership of liquidity providers in the shared pool. Accordingly, there is a need for the pool token contract implementation to follow the ERC20 specification. In the following, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic.

Our analysis shows that there is a minor ERC20 inconsistency or incompatibility issue found in the audited DODOv2. In particular, according to the ERC20 standard, `decimals()` is supposed to return `uint8`, instead of current `uint`.

In the following two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification.

Meanwhile, we notice in the `transferFrom()` routine, there is a common practice that is missing but widely used in other ERC20 contracts. Specifically, when `msg.sender = _from`, the current `transferFrom()` implementation disallows the token transfer if `msg.sender` has not explicitly allows

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

spending from herself yet. A common practice will whitelist this special case and allow `transferFrom()` if `msg.sender = _from` even there is no allowance specified. Also, if current allowance is the maximum `uint256`, there is no need to reduce the allowance as well.

```
    /**
     * @dev Transfer tokens from one address to another
     * @param from address The address which you want to send tokens from
     * @param to address The address which you want to transfer to
     * @param amount uint256 the amount of tokens to be transferred
     */
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public returns (bool) {
        require(amount <= _SHARES_[from], "BALANCE_NOT_ENOUGH");
        require(amount <= _ALLOWED_[from][msg.sender], "ALLOWANCE_NOT_ENOUGH");

        _SHARES_[from] = _SHARES_[from].sub(amount);
        _SHARES_[to] = _SHARES_[to].add(amount);
        _ALLOWED_[from][msg.sender] = _ALLOWED_[from][msg.sender].sub(amount);
        emit Transfer(from, to, amount);
        return true;
    }
```

Listing 3.10: DVMVault::transferFrom())

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approve() event	Is emitted on any successful call to approve()	✓

Recommendation Be compliant with the widely-accepted ERC20 specification and improve the `transferFrom()` logic by considering the special case when `msg.sender = _from`.

Status This issue has been partially fixed in `fc39f70`.

3.8 Trade Permission Bypass With Flashloan

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `DPPTrader`, `DVMTrader`
- Category: Security Features [8]
- CWE subcategory: CWE-269 [3]

Description

DODOv2 is designed to have a feature to turn on `whitelist` or `blacklistlist` (default). The default `blacklistlist` mode allows to block blacklisted traders from being involved in any trading operations with DODOv2; the `whitelist` mode allows traders only from the whitelisted traders. In the following, we show the associated modifiers that are defined to enforce the above mode. In current implementation, the `isSellAllow` modifier is attached to `sellBase()` and the `isBuyAllow` modifier is attached to `sellQuote()`.

```
    modifier isBuyAllow(address trader) {
        require(!_BUYING_CLOSE_ && _TRADE_PERMISSION_.isAllowed(trader), "
            TRADER_BUY_NOT_ALLOWED");
        _;
    }

    modifier isSellAllow(address trader) {
        require(
            !_SELLING_CLOSE_ && _TRADE_PERMISSION_.isAllowed(trader),
            "TRADER_SELL_NOT_ALLOWED"
        );
        _;
    }
```

Listing 3.11: DVMTrader.sol

In the meantime, we note that DODOv2 supports the `flashLoan()` feature that unfortunately can be exploited to bypass the above restriction.

```
    function flashLoan(
        uint256 baseAmount,
        uint256 quoteAmount,
        address assetTo,
        bytes calldata data
```

```
    ) external preventReentrant {
        _transferBaseOut(assetTo, baseAmount);
        _transferQuoteOut(assetTo, quoteAmount);

        if (data.length > 0)
            IDODOCallee(assetTo).DVMFlashLoanCall(msg.sender, baseAmount, quoteAmount, data);

        uint256 baseBalance = _BASE_TOKEN_.balanceOf(address(this));
        uint256 quoteBalance = _QUOTE_TOKEN_.balanceOf(address(this));

        // no input -> pure loss
        require(
            baseBalance >= _BASE_RESERVE_  quoteBalance >= _QUOTE_RESERVE_,
            "FLASH_LOAN_FAILED"
        );

        // sell quote
        if (baseBalance < _BASE_RESERVE_) {
            uint256 quoteInput = quoteBalance.sub(_QUOTE_RESERVE_);
            (uint256 receiveBaseAmount, uint256 mtFee) = querySellQuote(tx.origin, quoteInput);
            require(_BASE_RESERVE_.sub(baseBalance) <= receiveBaseAmount, "FLASH_LOAN_FAILED");

            _transferBaseOut(_MAINTAINER_, mtFee);
            emit DODOSwap(
                address(_QUOTE_TOKEN_),
                address(_BASE_TOKEN_),
                quoteInput,
                receiveBaseAmount,
                tx.origin
            );
        }

        // sell base
        if (quoteBalance < _QUOTE_RESERVE_) {
            uint256 baseInput = baseBalance.sub(_BASE_RESERVE_);
            (uint256 receiveQuoteAmount, uint256 mtFee) = querySellBase(tx.origin, baseInput);
            require(_QUOTE_RESERVE_.sub(quoteBalance) <= receiveQuoteAmount, "FLASH_LOAN_FAILED");

            _transferQuoteOut(_MAINTAINER_, mtFee);
            emit DODOSwap(
                address(_BASE_TOKEN_),
                address(_QUOTE_TOKEN_),
                baseInput,
                receiveQuoteAmount,
                tx.origin
            );
        }
```

```
        _sync();
    }
```

Listing 3.12: DVMTrader::flashLoan()

Specifically, `flashLoan()` implements a rather standard functionality in firstly transferring the requested loans to a designated recipient, then invoking a notification routine to the recipient, next checking the asset balance, and finally performing corresponding `base`/`quote`-selling operation. We point out that this routine does not properly trading permission that has been enforced in `sellBase()` and `sellQuote()`.

Recommendation Properly add validation checks in`flashLoan()` to enforce trading permissions based on either `whitelist` or `blacklistlist`. An example revision is shown below:

```
    function flashLoan(
        uint256 baseAmount,
        uint256 quoteAmount,
        address assetTo,
        bytes calldata data
    ) external preventReentrant {
        require(_TRADE_PERMISSION_.isAllowed(assetTo), "TRADER_BUY_NOT_ALLOWED");
        _transferBaseOut(assetTo, baseAmount);
        _transferQuoteOut(assetTo, quoteAmount);

        if (data.length > 0)
            IDODOCallee(assetTo).DVMFlashLoanCall(msg.sender, baseAmount, quoteAmount, data);

        uint256 baseBalance = _BASE_TOKEN_.balanceOf(address(this));
        uint256 quoteBalance = _QUOTE_TOKEN_.balanceOf(address(this));

        // no input -> pure loss
        require(
            baseBalance >= _BASE_RESERVE_  quoteBalance >= _QUOTE_RESERVE_,
            "FLASH_LOAN_FAILED"
        );

        // sell quote
        if (baseBalance < _BASE_RESERVE_) {
            require(!_BUYING_CLOSE_, "BUYING_NOT_ALLOWED");
            uint256 quoteInput = quoteBalance.sub(_QUOTE_RESERVE_);
            (uint256 receiveBaseAmount, uint256 mtFee) = querySellQuote(tx.origin, quoteInput);
            require(_BASE_RESERVE_.sub(baseBalance) <= receiveBaseAmount, "FLASH_LOAN_FAILED");

            _transferBaseOut(_MAINTAINER_, mtFee);
            emit DODOSwap(
                address(_QUOTE_TOKEN_),
                address(_BASE_TOKEN_),
```

```
                quoteInput,
                receiveBaseAmount,
                tx.origin
            );
        }

        // sell base
        if (quoteBalance < _QUOTE_RESERVE_) {
            require(!_SELLING_CLOSE_, "SELLING_NOT_ALLOWED");
            uint256 baseInput = baseBalance.sub(_BASE_RESERVE_);
            (uint256 receiveQuoteAmount, uint256 mtFee) = querySellBase(tx.origin,
                baseInput);
            require(_QUOTE_RESERVE_.sub(quoteBalance) <= receiveQuoteAmount, "
                FLASH_LOAN_FAILED");

            _transferQuoteOut(_MAINTAINER_, mtFee);
            emit DODOSwap(
                address(_BASE_TOKEN_),
                address(_QUOTE_TOKEN_),
                baseInput,
                receiveQuoteAmount,
                tx.origin
            );
        }

        _sync();
    }
```

Listing 3.13: Revised DVMTrader::flashLoan()

Status The issue has been fixed in this commit: `fc39f70`.

3.9 Confused Deputy For Fund-Stealing

- ID: PVE-009
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `DODOV1Proxy01`, `DODOV2Proxy01`
- Category: Security Features [8]
- CWE subcategory: CWE-269 [3]

Description

DODOv2 shares a similar approach in separating the swap-related core functionality from the wrapper functionality. The wrapper functionality provides transparent support of `Ether`, the native token on Ethereum. While reviewing the wrapper functionality, we notice a `DODOV2Proxy01::externalSwap()` routine. As the name indicates, this routine is designed to enable external swap integration with other similar DEX offerings.

However, our analysis shows that this routine can be exploited to abuse the trading users' trust on the privileged contract, i.e., `DODOApprove` to launch a so-called `confused deputy` attack. The consequence of this attack is to directly move funds from these trading users to attacker's account.

```
    function externalSwap(
        address fromToken,
        address toToken,
        address approveTarget,
        address to,
        uint256 fromTokenAmount,
        uint256 minReturnAmount,
        bytes memory callDataConcat,
        uint256 deadLine
    )
        external
        virtual
        override
        payable
        judgeExpired(deadLine)
        returns (uint256 returnAmount)
    {
        uint256 toTokenOriginBalance = IERC20(toToken).universalBalanceOf(msg.sender);

        if (fromToken != _ETH_ADDRESS_) {
            IDODOApprove(_DODO_APPROVE_).claimTokens(
                fromToken,
                msg.sender,
                address(this),
                fromTokenAmount
            );
            IERC20(fromToken).universalApproveMax(approveTarget, fromTokenAmount);
        }

        (bool success, ) = to.call{value: fromToken == _ETH_ADDRESS_ ? msg.value : 0}(
            callDataConcat);

        require(success, "DODOV2Proxy01: Contract Swap execution Failed");

        IERC20(fromToken).universalTransfer(
            msg.sender,
            IERC20(fromToken).universalBalanceOf(address(this))
        );

        IERC20(toToken).universalTransfer(
            msg.sender,
            IERC20(toToken).universalBalanceOf(address(this))
        );

        returnAmount = IERC20(toToken).universalBalanceOf(msg.sender).sub(
            toTokenOriginBalance);
        require(returnAmount >= minReturnAmount, "DODOV2Proxy01: Return amount is not
            enough");
```

```
473
474         emit OrderHistory(
475             fromToken,
476             toToken,
477             msg.sender,
478             fromTokenAmount,
479             returnAmount
480         );
481     }
```

Listing 3.14: DODOV2Proxy01::externalSwap()

Specifically, the issue lies in the external call at line 457: `to.call(callDataConcat)`. As both `to` and `callDataConcat` are part of input that should not be considered trustworthy, a malicious actor can craft an input by specifying `to=IDODOApprove(_DODO_APPROVE_)` and `callDataConcat` to invoke `_DODO_APPROVE_.claimTokens(fromToken, victim, attacker, amount)`. Since the `victim` trusts `IDODOApprove(_DODO_APPROVE_)`, her funds can be transferred to the attacker's account up to the permitted allowance.

The same issue is also applicable to `DODOV1Proxy01::externalSwap()`.

Recommendation Validate the given inputs and ensures `to != IDODOApprove(_DODO_APPROVE_)`.

Status The issue has been fixed in this commit: `fc39f70`.

4 Conclusion

In this audit, we have analyzed the DODOv2 documentation and implementation. The audited system presents a unique innovation and we are impressed by the overall design and solid implementation. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that 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-269: Improper Privilege Management. https://cwe.mitre.org/data/definitions/269.html.

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

[5] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[6] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

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

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

[9] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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