SMART CONTRACT AUDIT REPORT

for

OliveDAO

Prepared By: Patrick Lou

PeckShield

May 2, 2022

Document Properties

Client	Olive DAO
Title	Smart Contract Audit Report
Target	OliveDAO
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Patrick Lou, Jing Wang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 2, 2022	Xuxian Jiang	Final Release
1.0-rc	April 6, 2022	Xuxian Jiang	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction 4

1.1 About OliveDAO . . . 4

1.2 About PeckShield . . . 5

1.3 Methodology . . . 5

1.4 Disclaimer . . . 7

2 Findings 9

2.1 Summary . . . 9

2.2 Key Findings . . . 10

3 Detailed Results 11

3.1 Improper Funding Source in Locker::_deposit_for() . . . 11

3.2 Proper Pool/Manager Initialization in Pool/Manager::initialize() . . . 13

3.3 Trust Issue of Admin Keys . . . 14

3.4 Accommodation of Non-ERC20-Compliant Tokens . . . 15

4 Conclusion 18

References 19

1 Introduction

Given the opportunity to review the design document and related source code of the governance support in the `OliveDAO` 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 is well engineered, though it can be further improved by addressing the issues in this report. This document outlines our audit results.

1.1 About OliveDAO

`OliveDAO` enables users to provide liquidity without bearing the risk of impermanence loss while providing `web3` projects with an everlasting source of sustainable, low-cost liquidity for their project tokens. The audited governance support allows the utility token `OLIVE` holders to vote-lock and gain their `veOLIVE` so that they can participate in gauge voting, liquidity direction and governance, as well as receive corresponding rewards. By capturing fees and holding protocol tokens in its reserve, the protocol will build over time into a strong reserve of various assets in the `OLIVE` ecosystem.

The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of the audited protocol

Item	Description
Name	Olive DAO
Website	https://olivedao.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 2, 2022

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

- https://github.com/0xPolysynth/audit-contracts.git (28e80de)

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

- https://github.com/0xPolysynth/audit-contracts.git (ef49048)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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.

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

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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) [7], 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 1.4: 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 implementation of the governance support in the `OliveDAO` protocol. 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	2	
Low	2	
Informational	0	
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 that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings of OliveDAO Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Funding Source in Locker::_deposit_for()	Business Logic	Resolved
PVE-002	Low	Proper Pool/Manager Initialization in Pool/Manager::initialize()	Coding Practices	Resolved
PVE-003	Medium	Trust on Admin Keys	Security Features	Confirmed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improper Funding Source in Locker::_deposit_for()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Locker
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

By design, the only way to obtain the governance veOLIVE tokens is by locking OLIVE tokens. A user's veOLIVE balance decays linearly over the time. And the rewards are distributed weekly and proportionally to veOLIVE holders's balance. While reviewing the current locking logic, we notice the key helper routine _deposit_for() needs to be revised.

To elaborate, we show below the implementation of this _deposit_for() helper routine. In fact, it is an internal function to perform deposit and lock OLIVE for a user. This routine has a number of arguments and the first one _addr is the address to receive the veOLIVE balance. It comes to our attention that the _addr address is also the one to actually provide the assets, `oliveToken.transferFrom(_addr, address(this), _value))` (line 255). In fact, the `msg.sender` should be the one to provide the assets for locking! Otherwise, this function may be abused to lock veOLIVE tokens from users who have approved the locking contract before without their notice.

```
231    function _deposit_for(address _addr, uint256 _value, uint256 unlock_time,
           LockedBalance memory locked_balance, int128 vetype) internal {
232        LockedBalance memory old_locked;

234        LockedBalance memory _locked = locked_balance;
235        uint256 supply_before = supply;

237        supply = supply_before + _value;
238        old_locked.amount = _locked.amount;
239        old_locked.end = _locked.end;
```

```
        // Adding to existing lock, or if a lock is expired - creating a new one
        _locked.amount += uint256Toint128(_value);
        if (unlock_time != 0) {
            _locked.end = unlock_time;
        }
        locked[_addr] = _locked;

        // Possibilities:
        // Both old_locked.end could be current or expired (>/< _blockTimestamp())
        // value == 0 (extend lock) or value > 0 (add to lock or extend lock)
        // _locked.end > _blockTimestamp() (always)
        _checkpoint(_addr, old_locked, _locked);

        if (_value != 0) {
            require(oliveToken.transferFrom(_addr, address(this), _value), "Payment
                error");
        }

        uint256 totalVeOlive = int128Touint256(user_point_history[_addr][epoch].bias);
        uint256 _epoch = epoch;
        Point memory last_point = point_history[_epoch];
        uint256 deltaVeOlive = int128Touint256(last_point.bias);

        emit Deposit(_addr, _value, _locked.end, vetype, _blockTimestamp(), totalVeOlive
            , deltaVeOlive);
        emit Supply(supply_before, supply_before + _value);
    }
```

Listing 3.1: Locker::_deposit_for()

Recommendation Revise the above helper routine to use the right funding source to transfer the assets for locking.

Status The issue has been fixed in the following commit: ef49048.

3.2 Proper Pool/Manager Initialization in Pool/Manager::initialize()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pool`, `Manager`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The governance support also comes with core `Pool` and `Manager` contracts. These contracts inherit from a number of well-defined modules, such as `ERC20`, `Ownable`, and `ReentrancyGuard`. Note these parent contracts also come with their own specific initialization routines. While examining the initialization logic of these two contracts (`Pool` and `Manager`), we notice their current implementation needs to improved.

To elaborate, we show below the implementation of the `initialize()` helper routine from the `Pool` contract. It has properly invoked the initialization routines from the inherited `ERC20`, `Ownable`, and `Pausable`. However, it comes to our attention the initialization routine from the inherited `ReentrancyGuard` is not invoked. The lack of the `ReentrancyGuard` initialization may cause issues in the needed reentrancy protection.

```
    function initialize(
        ERC20 _underlyer,
        IManager _manager,
        string memory name_,
        string memory symbol_
    ) public initializer {
        require(address(_underlyer) != address(0), "ZERO_ADDRESS");
        require(address(_manager) != address(0), "ZERO_ADDRESS");

        __Context_init_unchained();
        __Ownable_init_unchained();
        __Pausable_init_unchained();
        __ERC20_init_unchained(name_, symbol_);

        underlyer = _underlyer;
        manager = _manager;
    }
```

Listing 3.2: Pool:: initialize ()

Note the same issue is also applicable to the `Manager` contract.

Recommendation Revise the above helper routine to properly initialize the inherited contracts.

Status The issue has been fixed in the following commit: ef49048.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `OliveDAO` protocol, there are certain privileged accounts, i.e., `admin`. When examining the related contracts, we notice an inherent trust on these privileged accounts. For example, this `admin` account plays a critical role in governing and regulating the system-wide operations (e.g., configure various settings). It also has the privilege to control or govern the flow of assets within the protocol contracts . In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function registerController(bytes32 id, address controller) external override
        onlyAdmin {
        require(controllerIds.add(id), "CONTROLLER_EXISTS");
        registeredControllers[id] = controller;
        emit ControllerRegistered(id, controller);
    }

    function unRegisterController(bytes32 id) external override onlyAdmin {
        require(controllerIds.remove(id), "INVALID_CONTROLLER");
        delete registeredControllers[id];
        emit ControllerUnregistered(id, registeredControllers[id]);
    }

    function registerPool(address pool) external override onlyAdmin {
        require(pools.add(pool), "POOL_EXISTS");
        emit PoolRegistered(pool);
    }

    function unRegisterPool(address pool) external override onlyAdmin {
        require(pools.remove(pool), "INVALID_POOL");
        emit PoolUnregistered(pool);
    }

    function setVoting(address _voting) external override onlyAdmin {
        require(voting == address(0), "Already Initialized");
        require(_voting != address(0), "Cannot be zero address");
        voting = _voting;
        emit VotingSet(voting);
```

```
    }

    function setCycleDuration(uint256 duration) external override onlyAdmin {
        require(duration > 0, "Cannot be 0");
        cycleDuration = duration;
        emit CycleDurationSet(duration);
    }
```

Listing 3.3: Example Privileged Operations in `Manager`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the `admin` 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 `admin` explicit to the protocol 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: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [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 `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
```

```
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.4: USDT Token **Contract**

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

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.5: `SafeERC20::safeApprove()`

In current implementation, if we examine the `Pool::approveManager()` routine, it is designed to authorize the given account for the intended spending. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `safeIncreaseAllowance()` (lines 138 and 141). Moreover, the `safeApprove()` call needs to be invoked twice: the first time resets the allowance to 0 and the second time sets the intended allowance amount.

```
    function approveManager(uint256 amount) public override onlyOwner {
        uint256 currentAllowance = underlyer.allowance(address(this), address(manager));
        if (currentAllowance < amount) {
            uint256 delta = amount - currentAllowance;
            underlyer.safeIncreaseAllowance(address(manager), delta);
        } else {
            uint256 delta = currentAllowance - amount;
            underlyer.safeDecreaseAllowance(address(manager), delta);
        }
    }
```

Listing 3.6: `Pool::approveManager()`

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

Status The issue has been fixed in the following commit: ef49048.

4 Conclusion

In this audit, we have analyzed the design and implementation of the governance support in the OliveDAO protocol, which enables users to provide liquidity without bearing the risk of impermanence loss while providing web3 projects with an everlasting source of sustainable, low-cost liquidity for their project tokens. The current code base is well organized and 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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