SMART CONTRACT AUDIT REPORT

for

AladdinDAO

Prepared By: Yiqun Chen

PeckShield

August 14, 2021

Document Properties

Client	AladdinDAO
Title	Smart Contract Audit Report
Target	AladdinDAO
Version	1.1
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang, Shulin Bie
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.1	August 14, 2021	Xuxian Jiang	Final Version (Amended #1)
1.0	July 30, 2021	Xuxian Jiang	Final Version
1.0-rc1	July 18, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the AladdinDAO protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About AladdinDAO

AladdinDAO is a decentralized asset management protocol which shifts crypto investment from venture capitalists to wisdom of crowd. AladdinDAO aims to be the liquidity gateway for DeFi world by identifying and providing liquidity support to the most promising DeFi projects, and benefiting Aladdin and DeFi community from enjoying the fast growth and returns from selected projects. As a result, the protocol will help to reduce market information asymmetry and optimize asset and resources allocations for DeFi community overall. The audited upgrade improves the staking logic with the interaction with TokenMaster. The basic information of the AladdinDAO protocol is as follows:

Table 1.1: Basic Information of The AladdinDAO Protocol

Item	Description
Issuer	AladdinDAO
Website	https://aladdin.club/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 14, 2021

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

- https://github.com/AladdinDAO/aladdin-core.git (a5d68f5)

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

- https://github.com/AladdinDAO/aladdin-core.git (213f5c1)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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) [5], 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 design and implementation of the `AladdinDAO` 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	1	🟦
Low	3	🟦🟦🟦
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 1 medium-severity vulnerability, and 3 low-severity vulnerabilities.

Table 2.1: Key AladdinDAO Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-002	Medium	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-003	Low	Incorrect Pending ALD Reward Calculation	Business Logic	Fixed
PVE-004	Low	Improved Logic in BaseStrategy::withdraw()	Business Logic	Fixed

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 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Time and State [4]
- CWE subcategory: CWE-663 [1]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [9] exploit, and the recent `Uniswap/Lendf.Me` hack [8].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `TokenMaster` as an example, the `emergencyWithdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 208) starts before effecting the update on internal states (lines $210-211$), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
203      // Withdraw without caring about rewards. EMERGENCY ONLY.
204      function emergencyWithdraw(address _token) onlyValidPool(_token) external {
205          uint pid = tokenToPid[_token];
```

```
        PoolInfo storage pool = poolInfo[pid - 1];
        UserInfo storage user = userInfo[pid][msg.sender];
        IERC20(pool.token).safeTransfer(address(msg.sender), user.amount);
        emit EmergencyWithdraw(msg.sender, _token, user.amount);
        user.amount = 0;
        user.rewardDebt = 0;
    }
```

Listing 3.1: `TokenMaster::emergencyWithdraw()`

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`. However, it is important to take precautions in making use of `nonReentrant` to block possible `re-entrancy`. Note this suggestion is also applicable to other routines, including `deposit()` and `withdraw()` in the `BaseVault` contract.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been fixed by this commit: `e6fd0ed`.

3.2 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `TokenMaster`
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

The `AladdinDAO` protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    // Update the given pool's ALD allocation point. Can only be called by the owner.
    function set(address _token, uint256 _allocPoint, bool _withUpdate) external
        onlyOwner onlyValidPool(_token){
        if (_withUpdate) {
            massUpdatePools();
        }
        uint pid = tokenToPid[_token];
        totalAllocPoint = totalAllocPoint.sub(poolInfo[pid - 1].allocPoint).add(
            _allocPoint);
        poolInfo[pid - 1].allocPoint = _allocPoint;
    }
```

Listing 3.2: `TokenMaster::set()`

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

Status The issue has been fixed by this commit: `7ab8a4a`.

3.3 Incorrect Pending ALD Reward Calculation

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TokenMaster`
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

As mentioned in Section 3.2, the `AladdinDAO` protocol provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its `allocPoint *100%/totalAllocPoint` share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool. And the protocol provides a helper routine `pendingALD()` to query the pending ALD rewards.

To elaborate, we show below the full implementation of this helper routine. This routine properly computes the new overall `aldReward`, but fails to take into consideration the `tokenDistributor` portion. In other words, the portion of rewards that can be applied to the accumulated reward per share is smaller, i.e., `aldReward.sub(aldReward.mul(tokenDistributorAllocNume).div(tokenDistributorAllocDenom`

`))`. While the current logic may not rely on this helper routine, the front-end display of the protocol rewards to users may be misleading.

```
    // View function to see pending ALDs on frontend.
    function pendingALD(address _token, address _user) onlyValidPool(_token) external
        view returns (uint256) {
        uint pid = tokenToPid[_token];
        PoolInfo storage pool = poolInfo[pid - 1];
        UserInfo storage user = userInfo[pid][_user];
        uint256 accALDPerShare = pool.accALDPerShare;
        uint256 lpSupply = IERC20(pool.token).balanceOf(address(this));
        if (block.number > pool.lastRewardBlock && lpSupply != 0) {
            uint256 aldReward = aldPerBlock.mul(block.number.sub(pool.lastRewardBlock))
                                              .mul(pool.allocPoint)
                                              .div(totalAllocPoint);
            accALDPerShare = accALDPerShare.add(aldReward.mul(1e18).div(lpSupply));
        }
        return user.amount.mul(accALDPerShare).div(1e18).sub(user.rewardDebt);
    }
```

Listing 3.3: `TokenMaster::pendingALD()`

Moreover, it is important to emphasize that the current implementation does not support deflationary tokens as the pool tokens. With that, it is important to vet the pool tokens before they are added so that no deflationary tokens will be accidentally introduced into the protocol.

Recommendation Correct the above `pendingALD()` function by subtracting the `tokenDistributor` portion.

Status The issue has been fixed by this commit: `21e8178`.

3.4 Improved Logic in BaseStrategy::withdraw()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `BaseStrategy`
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

`AladdinDAO` is a decentralized asset management protocol which shifts crypto investment from venture capitalists to wisdom of crowd. The investment subsystem is inspired from the `yearn.finance` framework and thus shares similar architecture with `vaults`, `controller`, and `strategies`.

While examining the `strategy` implementation (inside the `BaseStrategy` contract), we notice a potential issue that needs to be addressed. Specifically, new `strategy` contracts of `Aladdin` have been designed and implemented to invest VC assets (held in `vaults`), harvest growing yields, and return any gains, if any, to the investors. In order to collect stuck assets in these `strategy` contracts, each `strategy` implements a dedicated function, i.e., `withdraw()`, that can be invoked to collect non-`want` assets back to the controller, which further passes through to the governance.

```
    // Controller only function for creating additional rewards from dust
    function withdraw(IERC20 _asset) external returns (uint balance) {
        require(msg.sender == controller, "!controller");
        require(want != address(_asset), "want");
        balance = _asset.balanceOf(address(this));
        _asset.safeTransfer(controller, balance);
    }
```

Listing 3.4: `BaseStrategy::withdraw()`

To elaborate, we show above this `withdraw()` function. It comes to our attention that it properly excludes the `want` assets from being collected. However, it does not exclude the `reward` assets.

Recommendation Improved the above `withdraw()` function by also excluding the `reward` token. An example revision is shown below.

```
    // Controller only function for creating additional rewards from dust
    function withdraw(IERC20 _asset) external returns (uint balance) {
        require(msg.sender == controller, "!controller");
        require(want != address(_asset) && reward != address(_asset), "want");
        balance = _asset.balanceOf(address(this));
        _asset.safeTransfer(controller, balance);
    }
```

Listing 3.5: `BaseStrategy::withdraw()`

Status The issue has been fixed by this commit: `d726dd2`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `AladdinDAO` protocol. The audited system presents a unique addition to current DeFi offerings by offering a decentralized asset management protocol which shifts crypto investment from venture capitalists to wisdom of crowd. 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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

[4] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[8] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[9] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.