SMART CONTRACT AUDIT REPORT

for

AladdinDAOv2

Prepared By: Yiqun Chen

PeckShield

December 21, 2021

Document Properties

Client	AladdinDAO
Title	Smart Contract Audit Report
Target	AladdinDAO
Version	1.0
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
1.0	December 21, 2021	Xuxian Jiang	Final Release
1.0-rc	December 18, 2021	Xuxian Jiang	Release Candidate)

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the upgraded `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 provides new vaults as well as new bonding and reward support. The basic information of the audited 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	December 21, 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-contracts.git (a2490d8)

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-contracts.git (f446af3)

1.2 About PeckShield

PeckShield Inc. [11] 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

	Likelihood		
Impact	*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 [10]:

- 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) [9], 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 the upgraded 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	1	🟦
Medium	2	🟦🟦
Low	2	🟦🟦
Informational	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities 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 high-severity vulnerability, 2 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key AladdinDAO Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper Reward Token Validation in MultipleRewardsVaultBase	Business Logic	Fixed
PVE-002	Medium	Proper Normalization in UniswapV2PriceOracle	Business Logic	Fixed
PVE-003	High	Price Manipulation in UniswapV2PairPriceOracle	Time and State	Fixed
PVE-004	Medium	Trust on Admin Keys	Security Features	Mitigated
PVE-005	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	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 Proper Reward Token Validation in MultipleRewardsVaultBase

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MultipleRewardsVaultBase`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1099 [1]

Description

The upgraded `AladdinDAO` protocol supports new vaults as well as unique reward mechanisms. Each vault may support a number of reward tokens, which are used to reward protocol users. While reviewing the current support, we notice an inherent restriction on the maximum number of supported reward tokens and this restriction needs to be honored in current vault contracts.

In particular, the `RewardBondDepositor` contract defines a constant `MAX_REWARD_TOKENS`, which restricts the maximum number of reward tokens for each vault to be 4. However, while reviewing current vaults, in particular `MultipleRewardsVaultBase`, it comes to our attention this restriction is currently not recognized and enforced.

```
/// @dev setup reward tokens, should be called in constructor.
/// @param _rewardTokens A list of reward tokens.
function _setupRewardTokens(address[] memory _rewardTokens) internal {
  rewardTokens = _rewardTokens;
  for (uint256 i = 0; i < _rewardTokens.length; i++) {
    IERC20(_rewardTokens[i]).safeApprove(depositor, uint256(-1));
  }
}
```

Listing 3.1: `MultipleRewardsVaultBase::_setupRewardTokens()`

We need to mention that the current vaults do not have more than 4 reward tokens. However, this needs to be better enforced at the smart contract implementation. A candidate function for the needed enforcement is the above _setupRewardTokens() function.

Recommendation Enforce the maximum number of reward tokens in each supported vault.

Status This issue has been fixed in the following commit: 45bf930.

3.2 Proper Normalization in UniswapV2PriceOracle

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: UniswapV2PriceOracle
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

Description

The AladdinDAO protocol introduces UniswapV2-based price oracles. To facilitate the price-based calculation, the current token price needs to be normalized with the 1e18 scale. However, our analysis shows the current normalization logic is flawed.

To elaborate, we show below the affected price() function from the UniswapV2PriceOracle contract. While it retrieves the right reserves, these reserves are not properly scaled to 1e18. For example, the current _reserve0 normalization of _reserve0.mul(10**IERC20Metadata(_token0).decimals()) (line 49) should be _reserve0.mul(1e18).div(10**IERC20Metadata(_token0).decimals()). And the _reserve1 counterpart (line 52) needs to be _reserve1.mul(1e18).div(10**IERC20Metadata(_token1).decimals()).

```
38  function price(address _asset) public view override returns (uint256) {
39    address _pair = pairs[_asset];
40    require(_pair != address(0), "UniswapV2PriceOracle: not supported");
41
42    uint256 _basePrice = IPriceOracle(chainlink).price(base);
43    (uint256 _reserve0, uint256 _reserve1, ) = IUniswapV2Pair(_pair).getReserves();
44    address _token0 = IUniswapV2Pair(_pair).token0();
45    address _token1 = IUniswapV2Pair(_pair).token1();
46
47    // make reserve with scale 1e18
48    if (IERC20Metadata(_token0).decimals() < 18) {
49      _reserve0 = _reserve0.mul(10**IERC20Metadata(_token0).decimals());
50    }
51    if (IERC20Metadata(_token1).decimals() < 18) {
52      _reserve1 = _reserve1.mul(10**IERC20Metadata(_token1).decimals());
53    }
54
55    if (_asset == _token0) {
56      return _basePrice.mul(_reserve1).div(_reserve0);
57    } else {
58      return _basePrice.mul(_reserve0).div(_reserve1);
```

```
    }
  }
```

Listing 3.2: `UniswapV2PriceOracle::price()`

Recommendation Apply the proper normalization of current reserves to compute the oracle prices.

Status This issue has been fixed in the following commit: `45bf930`.

3.3 Price Manipulation in UniswapV2PairPriceOracle

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: High
- Target: `UniswapV2PairPriceOracle`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

To facilitate the bonding and reward in `AladdinDAO`, there is a constant need of evaluating an asset's price. Accordingly, the protocol has provided helper routines to facilitate the asset pricing from various oracles.

```
  function price(address _pair) public view override returns (uint256) {
    address _token0 = IUniswapV2Pair(_pair).token0();
    address _token1 = IUniswapV2Pair(_pair).token1();
    address _ald = ald;

    require(_token0 == _ald  _token1 == _ald, "UniswapV2PairPriceOracle: not supported")
        ;

    (uint256 _reserve0, uint256 _reserve1, ) = IUniswapV2Pair(_pair).getReserves();
    uint256 _totalSupply = IUniswapV2Pair(_pair).totalSupply();

    if (_token0 == _ald) {
      uint256 _amount = uint256(1e18).mul(_reserve1).div(_totalSupply);
      return IPriceOracle(chainlink).value(_token1, _amount);
    } else {
      uint256 _amount = uint256(1e18).mul(_reserve0).div(_totalSupply);
      return IPriceOracle(chainlink).value(_token0, _amount);
    }
  }
```

Listing 3.3: `UniswapV2PairPriceOracle::price()`

To elaborate, we show above an example helper routine in `UniswapV2PairPriceOracle`. We notice the conversion is routed to `UniswapV2`-based DEXs and the related spot reserves are used to compute the price! Therefore, they are vulnerable to possible front-running attacks, resulting in possible loss for the token conversion.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of farming users.

Status This issue has been fixed in the following commit: `45bf930`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `AladdinDAO` protocol, the privileged owner account plays a critical role in governing and regulating the system-wide operations (e.g., `vault` addition, reward adjustment, and parameter setting). It also has the privilege to control or govern the flow of assets for investment or full withdrawal among the three components, i.e., `vault`, `treasury`, and `staking`.

With great privilege comes great responsibility. Our analysis shows that the governance account is indeed privileged. To elaborate, we show below the related functions.

```
function manage(address _token, uint256 _amount) external override {
  require(isReserveToken[_token]  isLiquidityToken[_token], "Treasury: not accepted");
  require(isReserveManager[msg.sender], "Treasury: not approved manager");

  IERC20(_token).safeTransfer(msg.sender, _amount);
  emit ReservesManaged(_token, msg.sender, _amount);
}

/// @dev mint ALD reward.
/// @param _recipient The address of to receive ALD token.
/// @param _amount The amount of token.
function mintRewards(address _recipient, uint256 _amount) external override {
  require(isRewardManager[msg.sender], "Treasury: not approved manager");

  IALD(ald).mint(_recipient, _amount);

  emit RewardsMinted(msg.sender, _recipient, _amount);
}
```

Listing 3.4: `Treasury::manage()/mintRewards()`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the `owner/governance` 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 Promptly transfer the `owner/governance` 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 a multi-sig account to regulate the governance/controller privileges. The multi-sig account is managed by 9 accounts from the `Aladdin` community.

3.5 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `ALDDaoV2`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

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 [13] exploit, and the recent Uniswap/Lendf.Me hack [12].

We notice there are occasions where the checks-effects-interactions principle is violated. Using the ALDDaoV2 as an example, the stake() 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. For example, the interaction with the external contracts (line 79) start before effecting the update on internal states (lines 81−82), 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.

```
76   function stake(uint256 _amount) external {
77     _updateReward(msg.sender);
78
79     IERC20(dao).safeTransferFrom(msg.sender, address(this), _amount);
80
81     totalShares = totalShares.add(_amount);
82     shares[msg.sender] = shares[msg.sender].add(_amount);
83
84     emit Stake(msg.sender, _amount);
85   }
```

Listing 3.5: ALDDaoV2::stake()

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 to thwart possible re-entrancy.

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions principle and utilizing the necessary nonReentrant modifier to block possible re-entrancy.

Status This issue has been fixed in the following commit: 45bf930.

4 Conclusion

In this audit, we have analyzed the design and implementation of the upgraded AladdinDAO protocol, which provides new vaults as well as new bonding and reward support. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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

References

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

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

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

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

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

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

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

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

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

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