# SMART CONTRACT AUDIT REPORT

for

# AladdinDAO

Prepared By: Shuxiao Wang

PeckShield

April 27, 2021

## Document Properties

| | |
|---|---|
| Client | AladdinDAO |
| Title | Smart Contract Audit Report |
| Target | AladdinDAO |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Huaguo Shi, Xuxian Jiang |
| Reviewed by | Shuxiao Wang |
| Approved by | Xuxian Jiang |
| Classification | Confidential |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | April 27, 2021 | Xuxian Jiang | Final Release |
| 1.0-rc2 | March 30, 2021 | Xuxian Jiang | Release Candidate #2 |
| 1.0-rc1 | March 28, 2021 | Xuxian Jiang | Release Candidate #1 |
| 0.3 | March 28, 2021 | Xuxian Jiang | Additional Findings #2 |
| 0.2 | March 24, 2021 | Xuxian Jiang | Additional Findings #1 |
| 0.1 | March 18, 2021 | 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 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 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 | April 27, 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 (0392773)

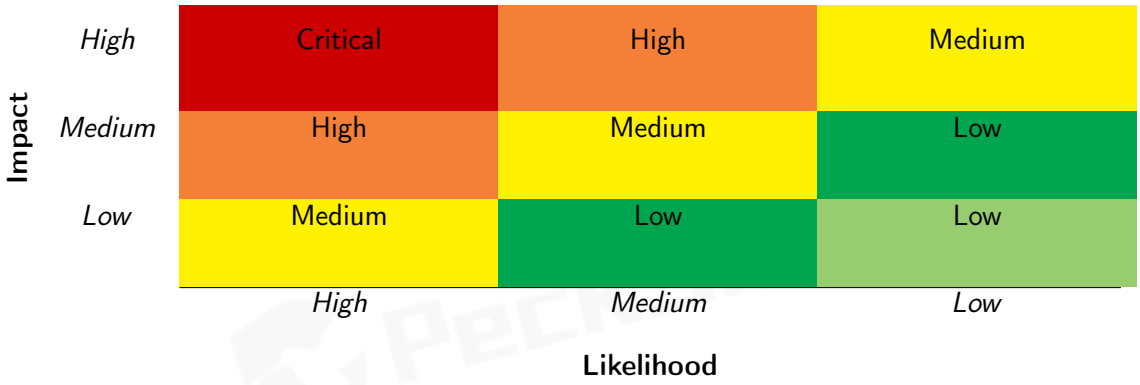
And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/AladdinDAO/aladdin-core.git (e960654)

## 1.2 About PeckShield

PeckShield Inc. [18] 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




## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [17]:

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

| Severity | # of Findings | |
|---|---|---|
| Critical | 0 | |
| High | 1 | ■ |
| Medium | 2 | ■■ |
| Low | 6 | ■■■■■■ |
| Informational | 3 | ■■■ |
| Total | 12 | |

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 6 low-severity vulnerabilities, and and 3 informational recommendations.

Table 2.1: Key AladdinDAO Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Force Investment Risk in BaseVault | Business Logic | Mitigated |
| PVE-002 | High | Possible Sandwich/MEV Attacks To Collect Most Rewards | Time and State | Fixed |
| PVE-003 | Informational | Potential Reentrancy Risk in TokenMaster | Time and State | Fixed |
| PVE-004 | Low | Possible Costly aldLPs From Improper Initialization | Time and State | Confirmed |
| PVE-005 | Low | Asset Consistency Check Between Vault And Strategy | Coding Practices | Fixed |
| PVE-006 | Informational | Recommended Explicit Pool Validity Checks | Security Features | Fixed |
| PVE-007 | Low | Duplicate Pool Detection and Prevention | Business Logic | Fixed |
| PVE-008 | Low | Timely massUpdatePools During Pool Weight Changes | Business Logic | Fixed |
| PVE-009 | Low | Improved Sanity Checks For System Parameters | Coding Practices | Fixed |
| PVE-010 | Medium | Inappropriate Initialization Of TokenMaster | Business Logic | Fixed |
| PVE-011 | Medium | Trust Issue Of Admin Keys | Security Features | Mitigated |
| PVE-012 | Informational | Redundant Code Removal | Coding Practices | 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 Force Investment Risk in BaseVault

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `BaseVault`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

### 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 `vault` implementation (inside the `BaseVault` contract), we notice a potential force investment risk that has been exploited in earlier hacks, e.g., `yDAI` [19] and `BT.Finance` [1]. To elaborate, we show blow the related `BaseVault::farm()` routine.

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 have a smooth investment experience, the `vault` contract has a dedicated function, i.e., `farm()`, that can be invoked to kick off the investment.

```
151   // Keepers call farm() to send funds to strategy
152   function farm() public {
153       uint _bal = available();
154
155       uint keeperFee = _bal.mul(farmKeeperFeeMin).div(max);
156       token.safeTransfer(msg.sender, keeperFee);
157
158       uint amountLessFee = _bal.sub(keeperFee);
159       token.safeTransfer(controller, amountLessFee);
160       IController(controller).farm(address(this), amountLessFee);
161
162       emit Farm(msg.sender, keeperFee, amountLessFee);
```

```
163   }
```

Listing 3.1: BaseVault::farm()

It comes to our attention that the `farm()` function is not guarded or can be invoked by any one to initiate the investment. If the configured strategy blindly invests the deposited funds into an imbalanced `Curve` pool, the strategy will not result in a profitable investment. In fact, earlier incidents (yDAI and BT hacks [19, 1]) have prompted the need of a guarded call to the `farm()` function. For the very same reason, we argue for the guarded call to `farm()` to block potential flashloan-assisted attacks. One mitigation will be to only allow for `EOA`-based trustworthy keepers.

**Recommendation** Ensure the `farm()` can only be called via a trusted entity. And take extra care in ensuring the vault assets will not be blindly deposited into a faulty strategy (that is currently not making any profit).

**Status** This issue has been confirmed. The team plans to address this issue by carefully choosing the LP tokens for farming so that the forced investment issue may be mitigated.

## 3.2 Possible Sandwich/MEV Attacks To Collect Most Rewards

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `BaseVault`
- Category: Time and State [15]
- CWE subcategory: CWE-682 [8]

### Description

As mentioned in Section 3.1, the `Aladdin` protocol has designed a new `strategy` contract to invest VC funds (held in `vaults`), harvest growing yields, and collect any gains, if any, to the share holders. In the meantime, we notice the `Aladdin` protocol takes a different approach by directly rewarding the yields back to investors.

To elaborate, we show below the `harvest()` function in `BaseVault`. This routine essentially collects any pending rewards via `controller::harvest()` (line 168) and then distributes the collected rewards evenly to current share holders (line 177).

```
165   // Keepers call harvest() to claim rewards from strategy
166   function harvest() external {
167       uint _rewardBefore = rewardToken.balanceOf(address(this));
168       IController(controller).harvest(address(this));
169       uint _rewardAfter = rewardToken.balanceOf(address(this));
170
171       uint harvested = _rewardAfter.sub(_rewardBefore);
```

```
172         uint keeperFee = harvested.mul(harvestKeeperFeeMin).div(max);
173         rewardToken.safeTransfer(msg.sender, keeperFee);
174
175         uint newRewardAmount = harvested.sub(keeperFee);
176         // distribute new rewards to current shares evenly
177         rewardsPerShareStored = rewardsPerShareStored.add(newRewardAmount.mul(1e18).div(
                totalSupply()));
178
179         emit Harvest(msg.sender, keeperFee, newRewardAmount);
180     }
```

Listing 3.2: BaseVault::harvest()

We notice the collected rewards are evenly distributed to share holders. With that, it is possible for a malicious actor to launch a flashloan-assisted deposit to claim the majority of rewards, resulting in significantly less rewards to legitimate share holders. This is possible as the `harvest()` routine is permissionless, allowing for a crafted contract to directly borrow a flashloan, deposit the borrowed loan into the vault pool, call `harvest()` to claim a majority share in rewards, and finally return the flashloan.

**Recommendation** Develop an effective mitigation to the above sandwich attack to better protect the interests of investors. An immediate mitigation will restrict the call to `harvest()` by `EOA` accounts or trusted keepers only.

**Status** The issue has been fixed by this commit: `6f93973`.

## 3.3 Potential Reentrancy Risk in TokenMaster

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `TokenMaster`
- Category: Time and State [14]
- CWE subcategory: CWE-663 [7]

### 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` [21] exploit, and the recent `Uniswap/Lendf.Me` hack [20].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `TokenMaster` as an example, the `deposit()` 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 168) starts before effecting the update on internal states (lines 169 and 171), 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 very same `deposit()` function.

```
156     // Deposit LP tokens to TokenMaster for ALD allocation.
157     function deposit(uint256 _pid, uint256 _amount) public {
158         PoolInfo storage pool = poolInfo[_pid];
159         UserInfo storage user = userInfo[_pid][msg.sender];
160         updatePool(_pid);
161         if (user.amount > 0) {
162             uint256 pending = user.amount.mul(pool.accALDPerShare).div(1e12).sub(user.
                    rewardDebt);
163             if(pending > 0) {
164                 safeALDTransfer(msg.sender, pending);
165             }
166         }
167         if(_amount > 0) {
168             pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
169             user.amount = user.amount.add(_amount);
170         }
171         user.rewardDebt = user.amount.mul(pool.accALDPerShare).div(1e12);
172         emit Deposit(msg.sender, _pid, _amount);
173     }
```

Listing 3.3: TokenMaster::deposit()

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`.

**Recommendation** Apply necessary reentrancy prevention by making use of the common `nonReentrant` modifier.

**Status** The issue has been fixed by this commit: `47e6533`.

## 3.4 Possible Costly aldLPs From Improper Initialization

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseVault`
- Category: Time and State [11]
- CWE subcategory: CWE-362 [5]

### Description

The `Aladdin` protocol allows users to deposit supported assets and get in return `ald`-wrapped tokens to represent the pool share. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool token, i.e., `aldUSDT`, extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine. This routine is used for participating users to deposit the supported assets (e.g., `USDT`) and get respective `aldUSDT` pool tokens in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
83  function deposit(uint _amount) external {
84      _updateReward(msg.sender);
85
86      uint _pool = balance();
87      token.safeTransferFrom(msg.sender, address(this), _amount);
88
89      uint shares = 0;
90      if (_pool == 0) {
91        shares = _amount;
92      } else {
93        shares = (_amount.mul(totalSupply())).div(_pool);
94      }
95      _mint(msg.sender, shares);
96      emit Deposit(msg.sender, _amount);
97  }
```

Listing 3.4: BaseVault::deposit()

Specifically, when the pool is being initialized (line 90), the share value directly takes the value of `_amount` (line 91), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shares = _amount = 1 WEI`. With that, the actor can further deposit a huge amount of `USDT` assets with the goal of making the `aldUSDT` pool token extremely expensive.

An extremely expensive `aldUSDT` 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 `deposit()` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

**Status** The issue has been confirmed. And the team decides to mitigate this issue by properly following a guarded launch process.

## 3.5 Asset Consistency Check Between Vault And Strategy

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Controller`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1099 [2]

### Description

In the `Aladdin` protocol, there is a one-to-one mapping between a `vault` and its `strategy`. To properly link a `vault` with its `strategy`, it is natural for the two to operate on the same underlying asset. For example, the `VaultUSDTCompound` allows for `USDT`-based deposits and withdraws. The associated `strategy`, i.e., a `StrategyCompoundUSDT`-based instance, naturally has `USDT` as the underlying asset. If these two have different underlying assets, the link should not be successful.

If we examine the `setStrategy()` routine in the `controller` contract, this routine allows for dynamic binding of the `vault` with a new `strategy` (line 98). A successful binding needs to satisfy a number of requirements. One specific example is shown as follows: `require(IVault(vaults[_token]).token()== Strategy(_strategy).want())`. Apparently, this requirement guarantees the consistency of the underlying asset between the `vault` and its associated `strategy`.

```
74    function setStrategy(address _vault, address _strategy) public {
75        require(msg.sender == governance, "!governance");
76
77        address _current = strategies[_vault];
78        if (_current != address(0)) {
```

```
79              IStrategy(_current).withdrawAll();
80          }
81          strategies[_vault] = _strategy;
82          vaults[_strategy] = _vault;
83      }
```

Listing 3.5: Controller :: setStrategy()

However, if we examine the `constructor()` of various `strategy` contracts (e.g., `StrategyCompoundUSDT` and `StrategySushiETHUSDC`), the requirement of having the same underlying asset is not enforced. A new `strategy` deployment with an ill-provided list of arguments with an unmatched underlying asset may cause unintended consequences, including possible asset loss. With that, we suggest to maintain an invariant by ensuring the consistency of the underlying asset when a new `strategy` is being deployed or linked.

**Recommendation** Ensure the consistency of the underlying asset between the `vault` and its associated `strategy`. An example revision is shown below.

```
74      function setStrategy(address _vault, address _strategy) public {
75          require(msg.sender == governance, "!governance");
76          require(IVault(vaults[_token]).token() == Strategy(_strategy).want(), "!asset")
77
78          address _current = strategies[_vault];
79          if (_current != address(0)) {
80              IStrategy(_current).withdrawAll();
81          }
82          strategies[_vault] = _strategy;
83          vaults[_strategy] = _vault;
84      }
```

Listing 3.6: Controller :: setStrategy()

**Status** The issue has been fixed by this commit: `44d1486`.

## 3.6 Recommended Explicit Pool Validity Checks

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `TokenMaster`
- Category: Security Features [10]
- CWE subcategory: CWE-287 [4]

### Description

The reward mechanism in AladdinDAO has a central contract – `TokenMaster` that has been tasked with the pool management, staking/unstaking support, as well as the reward distribution to various

pools and stakers. In the following, we show the key `pool` data structure. Note all added pools are maintained in an array `poolInfo`.

```
36    // Info of each pool.
37    struct PoolInfo {
38        IERC20 lpToken;           // Address of LP token contract.
39        uint256 allocPoint;       // How many allocation points assigned to this pool.
              ALDs to distribute per block.
40        uint256 lastRewardBlock;  // Last block number that ALDs distribution occurs.
41        uint256 accALDPerShare; // Accumulated ALDs per share, times 1e12. See below.
42    }
43    ...
44    // Info of each pool.
45    PoolInfo[] public poolInfo;
```

Listing 3.7: The PoolInfo Data Structure in TokenMaster

When there is a need to add a new pool, set a new `allocPoint` for an existing pool, stake (by depositing the supported assets), unstake (by redeeming previously deposited assets), query pending rewards, there is a constant need to perform sanity checks on the pool validity. The current implementation simply relies on the implicit, compiler-generated bound-checks of arrays to ensure the pool index stays within the array range `[0, poolInfo.length-1]`. However, considering the importance of validating given pools and their numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say `validatePool`. This new modifier essentially ensures the given_pool_id or _pid indeed points to a valid, live pool, and additionally give semantically meaningful information when it is not!

```
156   // Deposit LP tokens to TokenMaster for ALD allocation.
157   function deposit(uint256 _pid, uint256 _amount) public {
158       PoolInfo storage pool = poolInfo[_pid];
159       UserInfo storage user = userInfo[_pid][msg.sender];
160       updatePool(_pid);
161       if (user.amount > 0) {
162           uint256 pending = user.amount.mul(pool.accALDPerShare).div(1e12).sub(user.
                  rewardDebt);
163           if(pending > 0) {
164               safeALDTransfer(msg.sender, pending);
165           }
166       }
167       if(_amount > 0) {
168           pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
169           user.amount = user.amount.add(_amount);
170       }
171       user.rewardDebt = user.amount.mul(pool.accALDPerShare).div(1e12);
172       emit Deposit(msg.sender, _pid, _amount);
173   }
```

Listing 3.8: TokenMaster::deposit()

We highlight that there are a number of functions that can be benefited from the new pool-validating modifier, including `set()`, `deposit()`, `withdraw()`, `emergencyWithdraw()`, `pendingALD()` and `updatePool()`.

**Recommendation** Apply necessary sanity checks to ensure the given `_pid` is legitimate. Accordingly, a new modifier `validatePool` can be developed and appended to each function in the above list.

```
    modifier validatePool(uint256 _pid) {
        require(_pid < poolInfo.length, "chef: pool exists?");
        _;
    }
```

Listing 3.9: The New validatePool() Modifier

**Status** The issue has been fixed by this commit: `c7afd45`.

## 3.7 Duplicate Pool Detection and Prevention

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `TokenMaster, MultiStakingRewards`
- Category: Business Logics [13]
- CWE subcategory: CWE-841 [9]

### Description

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.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a proper governance procedure). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in `add()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
254     // Add a new lp to the pool. Can only be called by the owner.
255     // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you
            do.
256     function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public
            onlyOwner {
257         if (_withUpdate) {
258             massUpdatePools();
259         }
260         uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
261         totalAllocPoint = totalAllocPoint.add(_allocPoint);
262         poolInfo.push(PoolInfo({
263             lpToken: _lpToken,
264             allocPoint: _allocPoint,
265             lastRewardBlock: lastRewardBlock,
266             accALDPerShare: 0
267         }));
268     }
```

Listing 3.10: TokenMaster::add()

**Recommendation** Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
254     function checkPoolDuplicate(IERC20 _lpToken) public {
255         uint256 length = poolInfo.length;
256         for (uint256 pid = 0; pid < length; ++pid) {
257             require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
258         }
259     }
260
261     // Add a new lp to the pool. Can only be called by the owner.
262     // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you
            do.
263     function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public
            onlyOwner {
264         if (_withUpdate) {
265             massUpdatePools();
266         }
267         checkPoolDuplicate(_lpToken);
268         uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
269         totalAllocPoint = totalAllocPoint.add(_allocPoint);
270         poolInfo.push(PoolInfo({
271             lpToken: _lpToken,
272             allocPoint: _allocPoint,
273             lastRewardBlock: lastRewardBlock,
274             accALDPerShare: 0
275         }));
276     }
```

Listing 3.11: Revised TokenMaster::add()

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

**Status** The issue has been fixed by this commit: `79a13e6`.

## 3.8 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `TokenMaster`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

### Description

As mentioned in Section 3.7, 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.

```
270     // Update the given pool's ALD allocation point. Can only be called by the owner.
271     function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
272         if (_withUpdate) {
273             massUpdatePools();
274         }
275         totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
                );
276         poolInfo[_pid].allocPoint = _allocPoint;
277     }
```

Listing 3.12: 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.

```
270     // Update the given pool's ALD allocation point. Can only be called by the owner.
271     function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
272         massUpdatePools();
273         totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
                );
274         poolInfo[_pid].allocPoint = _allocPoint;
275     }
```

Listing 3.13: TokenMaster::set()

**Status** The issue has been fixed by this commit: a680c9c.

## 3.9 Improved Sanity Checks For System Parameters

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Controller, Strategies`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1126 [3]

### Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Aladdin` protocol is no exception. Specifically, if we examine the `BaseVault` contract, it has defined a number of protocol-wide risk parameters, e.g., `farmKeeperFeeMin` and `harvestKeeperFeeMin`. In the following, we show the corresponding routines that allow for their changes.

```
191   function setAvailableMin(uint _availableMin) external {
192       require(msg.sender == governance, "!governance");
193       availableMin = _availableMin;
194   }
195
196   function setFarmKeeperFeeMin(uint _farmKeeperFeeMin) external {
197       require(msg.sender == governance, "!governance");
198       farmKeeperFeeMin = _farmKeeperFeeMin;
199   }
200
201   function setHarvestKeeperFeeMin(uint _harvestKeeperFeeMin) external {
202       require(msg.sender == governance, "!governance");
203       harvestKeeperFeeMin = _harvestKeeperFeeMin;
204   }
```

Listing 3.14: A Number of Setters in BaseVault

Our result shows the update logic on these fee parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an

undesirable consequence. For example, an unlikely mis-configuration of a large fee parameter (say more than 100%) will revert the `farm()` operation. Also, we notice that the `setAvailableMin()` routine in the `BaseVault` contract can be improved by validating the given `_availableMin` argument. For example, we can at least ensure the following: `require (_availableMin<max)`.

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

**Status** The issue has been fixed by this commit: `f382ae9`.

## 3.10 Inappropriate Initialization Of TokenMaster

- ID: PVE-010
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `TokenMaster`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

### Description

In `AladdinDAO`, staking users may deposit their assets into the `TokenMaster` pool and get corresponding rewards in return. The `TokenMaster` pool has been designed to have several scheduled reward-release phases and each rewarding phase has a respective multiplier to boost the rewarded amount. When examining the reward-release schedules, we notice the schedules need to be revised to better fit into the intended goal.

```
97      constructor(
98          ALDToken _ALD,
99          address _tokenDistributor
100     ) public {
101         ALD = _ALD;
102         tokenDistributor = _tokenDistributor;
103         startBlock = block.number;

105         uint256 bonusEndBlock = startBlock.add(INITIAL_BONUS_BLOCKS);
106         CHANGE_MULTIPLIER_AT_BLOCK.push(bonusEndBlock);
107         for (uint256 i = 1; i < REWARD_MULTIPLIER.length - 1; i++) {
108             uint256 changeMultiplierAtBlock = bonusEndBlock.add(BLOCKS_PER_MULTIPLIER.
                    mul(i+1));
109             CHANGE_MULTIPLIER_AT_BLOCK.push(changeMultiplierAtBlock);
110         }
111         CHANGE_MULTIPLIER_AT_BLOCK.push(uint256(-1));
112     }
```

Listing 3.15: TokenMaster::**constructor**()

To elaborate, we show above the initialization logic implemented in the `constructor()` function. Note the variable `BLOCKS_PER_MULTIPLIER` defines the approximately one-year duration (in terms of blocks) per multiplier period. It comes to our attention that the first multiplier period ends at `bonusEndBlock` (line 105), but the second multiplier takes almost two years to finish (line 108). It turns out that the intended duration for the second multiplier period is one year, instead of current two years.

**Recommendation** Properly initialize the multiplier schedule. And an example revision is shown below:

```
97      constructor(
98          ALDToken _ALD,
99          address _tokenDistributor
100     ) public {
101         ALD = _ALD;
102         tokenDistributor = _tokenDistributor;
103         startBlock = block.number;

105         uint256 bonusEndBlock = startBlock.add(INITIAL_BONUS_BLOCKS);
106         CHANGE_MULTIPLIER_AT_BLOCK.push(bonusEndBlock);
107         for (uint256 i = 1; i < REWARD_MULTIPLIER.length - 1; i++) {
108             uint256 changeMultiplierAtBlock = bonusEndBlock.add(BLOCKS_PER_MULTIPLIER.
                    mul(i));
109             CHANGE_MULTIPLIER_AT_BLOCK.push(changeMultiplierAtBlock);
110         }
111         CHANGE_MULTIPLIER_AT_BLOCK.push(uint256(-1));
112     }
```

Listing 3.16: TokenMaster::**constructor**()

**Status** The issue has been fixed by this commit: `940c45d`.

## 3.11 Trust Issue of Admin Keys

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [10]
- CWE subcategory: CWE-287 [4]

### Description

In the `AladdinDAO` protocol, the privileged owner account plays a critical role in governing and regulating the system-wide operations (e.g., `vault`/`strategy` 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`, `controller`, and `strategy`.

With great privilege comes great responsibility. Our analysis shows that the governance account is indeed privileged. In the following, we examine the current privilege management graph in the `Aladdin` protocol (Figure 3.1).

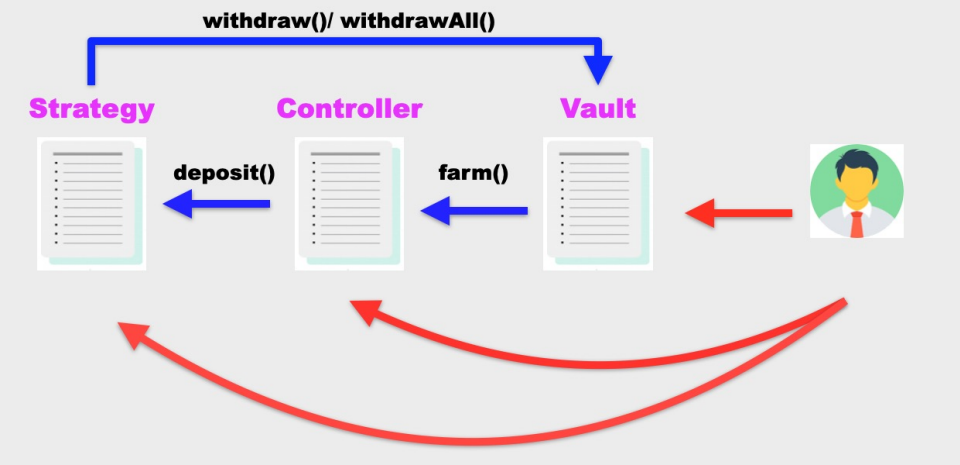



Figure 3.1: The Privilege Management Chain in `AladdinDAO`

We emphasize that the privilege assignment among `vault`, `controller`, and `strategy` is properly administrated. However, it is worrisome if the `governance` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the governance will be managed by a multi-sig account.

We point out that a compromised `governance` account would allow the attacker to add a malicious `controller` to steal all funds whenever the `farm()` call is made. It could also allow for the dynamic addition of a new malicious `strategy`, which directly undermines the assumption of the `Aladdin` protocol.

**Recommendation** Promptly transfer the `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.

## 3.12 Redundant Code Removal

- ID: PVE-012
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Controller`
- Category: Coding Practices [12]
- CWE subcategory: CWE-563 [6]

### Description

AladdinDAO makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the `Controller` contract has so far imported at least four reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the state variables defined in the `Controller` contract, two of them are not used: `split` and `max`. These unused states can be safely removed.

```
    uint public split = 500;
    uint public constant max = 10000;
```

Listing 3.17: Two Unused State Variables Defined in Controller

**Recommendation** Consider the removal of the redundant code with a simplified, consistent implementation.

**Status** The issue has been fixed by this commit: `194ee48`.

# 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] BT Finance. BT.Finance Exploit Analysis Report. https://btfinance.medium.com/bt-finance-exploit-analysis-report-a0843cb03b28.

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

[3] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

[8] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

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

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

[15] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

[19] PeckShield. The yDAI Incident Analysis: Forced Investment. https://peckshield.medium.com/the-ydai-incident-analysis-forced-investment-2b8ac6058eb5.

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

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