SMART CONTRACT AUDIT REPORT

for

AladdinDAO Concentrator

Prepared By: Xiaomi Huang

PeckShield

July 4, 2022

Document Properties

Client	AladdinDAO
Title	Smart Contract Audit Report
Target	AladdinDAO Concentrator
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 4, 2022	Xiaotao Wu	Final Release
1.0-rc	July 1, 2022	Xiaotao Wu	Release Candidate #1

Contact

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

Name	Xiaomi Huang
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 Concentrator 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 Concentrator

The AladdinDAO is a decentralized network to shift crypto investments from venture capitalists to the wisdom of crowds through collective value discovery. The Concentrator is a yield enhancement product by AladdinDAO built for farmers who wish to use their Convex LP assets to farm top-tier DeFi tokens (e.g., CRV, CVX) at the highest APY possible. The new version of the Concentrator adds an Initial Farming Offering (IFO) feature. During the IFO phase, the earnings of the Concentrator protocol will be converted into CTR tokens and distributed to users. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The AladdinDAO Concentrator

Item	Description
Name	AladdinDAO
Website	https://concentrator.aladdin.club/
Type	EVM Smart Contract
Platform	Solidity & Vyper
Audit Method	Whitebox
Latest Audit Report	July 4, 2022

In the following, we show the Git repository of reviewed files and the commit hash values used in this audit. Note the audited repository contains a number of sub-directories and this audit only covers the `concentrator` sub-directory and the `PlatformFeeDistributor/GaugeRewardDistributor` contracts in the `misc` sub-directory.

- https://github.com/AladdinDAO/aladdin-v3-contracts/tree/cont_vault (acc5007)

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

- https://github.com/AladdinDAO/aladdin-v3-contracts/tree/cont_vault (1340ea8)

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

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 [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.

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 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `AladdinDAO Concentrator` 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	4	■ ■ ■ ■
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 medium-severity vulnerability and 4 low-severity vulnerabilities.

Table 2.1: Key AladdinDAO Concentrator Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Costly _shares From Improper AladdinCRV Initialization	Time and State	Resolved
PVE-002	Low	Revisited Slippage Control In AladdinConvexVault::harvest()	Time and State	Mitigated
PVE-003	Low	Improved Logic In AladdinConvexVault::harvest()	Business Logic	Confirmed
PVE-004	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Possible Costly _shares From Improper AladdinCRV Initialization

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `AladdinCRV`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [3]

Description

The `AladdinDAO Concentrator` protocol allows users to deposit supported assets and get in return the share to represent the pool ownership. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool share 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 and get respective pool shares in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
  function _deposit(address _recipient, uint256 _amount) internal returns (uint256) {
    require(_amount > 0, "AladdinCRV: zero amount deposit");
    uint256 _underlying = totalUnderlying();
    uint256 _totalSupply = totalSupply();

    IERC20Upgradeable(CVXCRV).safeApprove(CVXCRV_STAKING, 0);
    IERC20Upgradeable(CVXCRV).safeApprove(CVXCRV_STAKING, _amount);
    IConvexBasicRewards(CVXCRV_STAKING).stake(_amount);

    uint256 _shares;
    if (_totalSupply == 0) {
      _shares = _amount;
    } else {
      _shares = _amount.mul(_totalSupply) / _underlying;
```

```
274         }
275         _mint(_recipient, _shares);

277         emit Deposit(msg.sender, _recipient, _amount);
278         return _shares;
279     }
```

Listing 3.1: `AladdinCRV::_deposit()`

Specifically, when the pool is being initialized (line 270), the share value directly takes the value of `_amount` (line 271), 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 call the `stakeFor()` function in the `CVXCRV_STAKING` contract to stake a huge amount of `CVXCRV` for the `AladdinCRV` contract with the goal of making the pool share extremely expensive.

An extremely expensive pool share can be very inconvenient to use as a small number of 1 `Wei` 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 resolved as the team confirms that `AladdinCRV` has already launched in mainnet.

3.2 Revisited Slippage Control In AladdinConvexVault::harvest()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AladdinConvexVault
- Category: Time and State [6]
- CWE subcategory: CWE-362 [3]

Description

The `AladdinConvexVault` contract of the `AladdinDAO Concentrator` protocol provides an external `harvest()` function to harvest the pending rewards from the `Convex` reward pool and convert the collected rewards to `aladdinCRV`, which will be distributed to the depositors. Users who call the `harvest()` function will receive a certain percentage of the converted `aladdinCRV` as harvest bounty. To facilitate the rewards convert, the helper routine `_swapCRVToCvxCRV` is called to swap `CRV` token to `CVXCRV` token (line 496).

```
467   function harvest(
468     uint256 _pid,
469     address _recipient,
470     uint256 _minimumOut
471   ) external virtual override onlyExistPool(_pid) nonReentrant returns (uint256
        harvested) {
472     PoolInfo storage _pool = poolInfo[_pid];
473     // 1. claim rewards
474     IConvexBasicRewards(_pool.crvRewards).getReward();
475
476     // 2. swap all rewards token to CRV
477     address[] memory _rewardsToken = _pool.convexRewardTokens;
478     uint256 _amount = address(this).balance;
479     address _token;
480     address _zap = zap;
481     for (uint256 i = 0; i < _rewardsToken.length; i++) {
482       _token = _rewardsToken[i];
483       if (_token != CRV) {
484         uint256 _balance = IERC20Upgradeable(_token).balanceOf(address(this));
485         if (_balance > 0) {
486           // saving gas
487           IERC20Upgradeable(_token).safeTransfer(_zap, _balance);
488           _amount = _amount.add(IZap(_zap).zap(_token, _balance, address(0), 0));
489         }
490       }
491     }
492     if (_amount > 0) {
493       IZap(_zap).zap{ value: _amount }(address(0), _amount, CRV, 0);
494     }
495     _amount = IERC20Upgradeable(CRV).balanceOf(address(this));
496     _amount = _swapCRVToCvxCRV(_amount, _minimumOut);
```

```
497
498         ...
499     }
```

Listing 3.2: AladdinConvexVault::harvest()

While examining the current swap logic, we notice it can be improved with effective slippage control. Specifically, although a user who calls the `harvest()` function has to specify the slippage control parameter (i.e., `_minimumOut`), this parameter specified by the user might be unreasonable. If this slippage control parameter is set to be 0, then there will be no slippage control in place, which opens up the possibility for front-running and potentially results in a smaller `CVXCRV` amount. Thus may cause loss for all depositors of the `AladdinConvexVault` contract. 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 liquidity provider.

Recommendation Develop an effective mitigation to the above sandwich arbitrage to better protect the interests of users.

Status This issue has been mitigated as the team confirms that they have inhouse harvest-bot that will call the `harvest()` function regularly.

3.3 Improved Logic In AladdinConvexVault::harvest()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AladdinConvexVault`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.2, the `harvest()` function in the `AladdinConvexVault` contract allows for users to harvest the pending rewards from the `Convex` reward pool and convert the collected rewards to `aladdinCRV`, which will be distributed to the depositors. While examining the routine, we notice the current implementation logic can be improved.

To elaborate, we show below its code snippet. It comes to our attention that if the `_pool.totalShare == 0`, there is no need to execute the harvest logic. Otherwise, the execution of the `_pool.accRewardPerShare.add(_rewards.mul(PRECISION)/ _pool.totalShare)` will revert (line 517).

```
    function harvest(
        uint256 _pid,
        address _recipient,
        uint256 _minimumOut
    ) external virtual override onlyExistPool(_pid) nonReentrant returns (uint256 harvested) {
        PoolInfo storage _pool = poolInfo[_pid];
        // 1. claim rewards
        IConvexBasicRewards(_pool.crvRewards).getReward();

        // 2. swap all rewards token to CRV
        address[] memory _rewardsToken = _pool.convexRewardTokens;
        uint256 _amount = address(this).balance;
        address _token;
        address _zap = zap;
        for (uint256 i = 0; i < _rewardsToken.length; i++) {
          _token = _rewardsToken[i];
          if (_token != CRV) {
            uint256 _balance = IERC20Upgradeable(_token).balanceOf(address(this));
            if (_balance > 0) {
              // saving gas
              IERC20Upgradeable(_token).safeTransfer(_zap, _balance);
              _amount = _amount.add(IZap(_zap).zap(_token, _balance, address(0), 0));
            }
          }
        }
        if (_amount > 0) {
          IZap(_zap).zap{ value: _amount }(address(0), _amount, CRV, 0);
        }
        _amount = IERC20Upgradeable(CRV).balanceOf(address(this));
        _amount = _swapCRVToCvxCRV(_amount, _minimumOut);

        _token = aladdinCRV; // gas saving
        _approve(CVXCRV, _token, _amount);
        uint256 _rewards = IAladdinCRV(_token).deposit(address(this), _amount);

        // 3. distribute rewards to platform and _recipient
        uint256 _platformFee = _pool.platformFeePercentage;
        uint256 _harvestBounty = _pool.harvestBountyPercentage;
        if (_platformFee > 0) {
          _platformFee = (_platformFee * _rewards) / FEE_DENOMINATOR;
          _rewards = _rewards - _platformFee;
          IERC20Upgradeable(_token).safeTransfer(platform, _platformFee);
        }
        if (_harvestBounty > 0) {
          _harvestBounty = (_harvestBounty * _rewards) / FEE_DENOMINATOR;
          _rewards = _rewards - _harvestBounty;
          IERC20Upgradeable(_token).safeTransfer(_recipient, _harvestBounty);
        }

        // 4. update rewards info
        _pool.accRewardPerShare = _pool.accRewardPerShare.add(_rewards.mul(PRECISION) /
```

```
            _pool.totalShare);

        emit Harvest(msg.sender, _rewards, _platformFee, _harvestBounty);

        return _amount;
    }
```

Listing 3.3: AladdinConvexVault::harvest()

Recommendation Only execute the harvest logic when `_pool.totalShare != 0`.

Status This issue has been confirmed.

3.4 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AddLiquidityHelper`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

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

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

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

```
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

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

Listing 3.4: `USDT Token Contract`

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `approve()` function does not have a return value. However, the `IERC20` interface has defined the following `approve()` interface with a `bool` return value: `function approve(address spender, uint256 amount)external returns (bool)`. As a result, the call to `approve()` may expect a return value. With the lack of return value of `USDT`'s `approve()`, the call will be unfortunately reverted.

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In the following, we use the `GaugeRewardDistributor::_distributeReward()` routine as an example. If the `USDT` token is supported as the reward token, the unsafe version of `IERC20(_token).approve()` (lines 296-297) may revert as there is no return value in the `USDT` token contract's `approve()` implementation (but the `IERC20` interface expects a return value)!

```
280    /// @dev Internal function to distribute reward to gauges.
281    /// @param _token The address of reward token.
282    /// @param _amount The amount of reward token.
283    function _distributeReward(address _token, uint256 _amount) internal {
284      RewardDistribution[] storage _distributions = distributions[_token];
285      uint256 _length = _distributions.length;
286      for (uint256 i = 0; i < _length; i++) {
287        RewardDistribution memory _distribution = _distributions[i];
288        if (_distribution.percentage > 0) {
289          uint256 _part = _amount.mul(_distribution.percentage) / FEE_DENOMINATOR;
290          GaugeRewards storage _gauge = gauges[_distribution.gauge];
291          if (_gauge.gaugeType == GaugeType.CurveGaugeV1V2V3) {
292            // @note Curve Gauge V1, V2 or V3 need explicit claim.
293            _gauge.pendings[_token] = _part.add(_gauge.pendings[_token]);
294          } else if (_gauge.gaugeType == GaugeType.CurveGaugeV4V5) {
295            // @note rewards can be deposited to Curve Gauge V4 or V5 directly.
296            IERC20(_token).approve(_distribution.gauge, 0);
297            IERC20(_token).approve(_distribution.gauge, _part);
298            ICurveGaugeV4V5(_distribution.gauge).deposit_reward_token(_token, _part);
299          } else {
300            // no gauge to distribute, just send to owner
301            IERC20(_token).safeTransfer(owner(), _part);
```

```
        }
      }
    }
  }
```

Listing 3.5: `GaugeRewardDistributor::_distributeReward()`

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

Status This issue has been fixed in the following commit: `1340ea8`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `AladdinDAO Concentrator` protocol, there are two privileged accounts, i.e., `owner` and `admin`. These accounts play a critical role in governing and regulating the system-wide operations (e.g., pause deposit/withdraw, set minter for `CTR` token, add/remove reward token, update key parameters, etc.). Our analysis shows that these privileged accounts need to be scrutinized. In the following, we use the `AladdinConvexVault` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

```
  /// @dev Update the withdraw fee percentage.
  /// @param _pid - The pool id.
  /// @param _feePercentage - The fee percentage to update.
  function updateWithdrawFeePercentage(uint256 _pid, uint256 _feePercentage) external
      onlyExistPool(_pid) onlyOwner {
    require(_feePercentage <= MAX_WITHDRAW_FEE, "fee too large");

    poolInfo[_pid].withdrawFeePercentage = _feePercentage;

    emit UpdateWithdrawalFeePercentage(_pid, _feePercentage);
  }

  /// @dev Update the platform fee percentage.
  /// @param _pid - The pool id.
  /// @param _feePercentage - The fee percentage to update.
  function updatePlatformFeePercentage(uint256 _pid, uint256 _feePercentage) external
      onlyExistPool(_pid) onlyOwner {
```

```
    require(_feePercentage <= MAX_PLATFORM_FEE, "fee too large");

    poolInfo[_pid].platformFeePercentage = _feePercentage;

    emit UpdatePlatformFeePercentage(_pid, _feePercentage);
  }

  /// @dev Update the harvest bounty percentage.
  /// @param _pid - The pool id.
  /// @param _percentage - The fee percentage to update.
  function updateHarvestBountyPercentage(uint256 _pid, uint256 _percentage) external
      onlyExistPool(_pid) onlyOwner {
    require(_percentage <= MAX_HARVEST_BOUNTY, "fee too large");

    poolInfo[_pid].harvestBountyPercentage = _percentage;

    emit UpdateHarvestBountyPercentage(_pid, _percentage);
  }

  /// @dev Update the recipient
  function updatePlatform(address _platform) external onlyOwner {
    require(_platform != address(0), "zero platform address");
    platform = _platform;

    emit UpdatePlatform(_platform);
  }

  /// @dev Update the zap contract
  function updateZap(address _zap) external onlyOwner {
    require(_zap != address(0), "zero zap address");
    zap = _zap;

    emit UpdateZap(_zap);
  }

  /// @dev Update the migrator contract
  function updateMigrator(address _migrator) external onlyOwner {
    require(_migrator != address(0), "zero migrator address");
    migrator = _migrator;

    emit UpdateMigrator(_migrator);
  }

  /// @dev Add new Convex pool.
  /// @param _convexPid - The Convex pool id.
  /// @param _rewardTokens - The list of addresses of reward tokens.
  /// @param _withdrawFeePercentage - The withdraw fee percentage of the pool.
  /// @param _platformFeePercentage - The platform fee percentage of the pool.
  /// @param _harvestBountyPercentage - The harvest bounty percentage of the pool.
  function addPool(
    uint256 _convexPid,
    address[] memory _rewardTokens,
```

```
        uint256 _withdrawFeePercentage,
        uint256 _platformFeePercentage,
        uint256 _harvestBountyPercentage
    ) external onlyOwner {
        for (uint256 i = 0; i < poolInfo.length; i++) {
            require(poolInfo[i].convexPoolId != _convexPid, "duplicate pool");
        }

        require(_withdrawFeePercentage <= MAX_WITHDRAW_FEE, "fee too large");
        require(_platformFeePercentage <= MAX_PLATFORM_FEE, "fee too large");
        require(_harvestBountyPercentage <= MAX_HARVEST_BOUNTY, "fee too large");

        IConvexBooster.PoolInfo memory _info = IConvexBooster(BOOSTER).poolInfo(_convexPid);
        poolInfo.push(
            PoolInfo({
                totalUnderlying: 0,
                totalShare: 0,
                accRewardPerShare: 0,
                convexPoolId: _convexPid,
                lpToken: _info.lptoken,
                crvRewards: _info.crvRewards,
                withdrawFeePercentage: _withdrawFeePercentage,
                platformFeePercentage: _platformFeePercentage,
                harvestBountyPercentage: _harvestBountyPercentage,
                pauseDeposit: false,
                pauseWithdraw: false,
                convexRewardTokens: _rewardTokens
            })
        );

        emit AddPool(poolInfo.length - 1, _convexPid, _rewardTokens);
    }

    /// @dev update reward tokens
    /// @param _pid - The pool id.
    /// @param _rewardTokens - The address list of new reward tokens.
    function updatePoolRewardTokens(uint256 _pid, address[] memory _rewardTokens) external onlyExistPool(_pid) onlyOwner {
        delete poolInfo[_pid].convexRewardTokens;
        poolInfo[_pid].convexRewardTokens = _rewardTokens;

        emit UpdatePoolRewardTokens(_pid, _rewardTokens);
    }

    /// @dev Pause withdraw for specific pool.
    /// @param _pid - The pool id.
    /// @param _status - The status to update.
    function pausePoolWithdraw(uint256 _pid, bool _status) external onlyExistPool(_pid) onlyOwner {
        poolInfo[_pid].pauseWithdraw = _status;

```

```
      emit PausePoolWithdraw(_pid, _status);
    }

    /// @dev Pause deposit for specific pool.
    /// @param _pid - The pool id.
    /// @param _status - The status to update.
    function pausePoolDeposit(uint256 _pid, bool _status) external onlyExistPool(_pid) onlyOwner {
      poolInfo[_pid].pauseDeposit = _status;

      emit PausePoolDeposit(_pid, _status);
    }
```

Listing 3.6: Example Privileged Operations in `AladdinConvexVault`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, 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 mitigated as the team confirms that the privileged `owner` account is a multi-sig.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `AladdinDAO Concentrator` protocol. The `Concentrator` is a yield enhancement product by `AladdinDAO` built for farmers who wish to use their `Convex LP` assets to farm top-tier `DeFi tokens` (e.g., `CRV`, `CVX`) at the highest `APY` possible. The new version of the `Concentrator` adds an `Initial Farming Offering` (IFO) feature. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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