Security Audit Report

Concentrator by AladdinDAO

SECBIT

March 4, 2022

1. Introduction

The AladdinDAO is a decentralized network to shift crypto investments from venture capitalists to the wisdom of crowds through collective value discovery. Concentrator is a yield enhancement product by AladdinDAO built for smart farmers who wish to use their Convex LP assets to farm top-tier DeFi tokens (CRV, CVX) at the highest APY possible. SECBIT Labs conducted an audit from February 22nd to March 4th, 2022, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the Concentrator contract has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising(see part 4 for details).

Type	Description	Level	Status
Implementation	4.3.1 The parameter `UserInfo.shares` is not recorded and updated, which directly results in the user not being able to retrieve the principal and rewards.	Medium	Fixed
Design & Implementation	4.3.2 If the function `_swapCRVToCvxCRV()` returns an incorrect value, the `harvest()` function may fail to execute.	Low	Fixed
Implementation	4.3.3 The function `_claim()` will duplicate the reward when processing aCRV token rewards.	Medium	Fixed
Design & Implementation	4.3.4 It may fail for the user to call the `depositWithCRV()` function to deposit the CRV token.	Low	Fixed
Design & Implementation	4.3.5 Potential loss of profits in `aladdinConvexVault` contract.	Info	Discussed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the Concentrator contract is shown below:

- Project website
  - https://aladdin.club/
- Smart contract code
  - https://github.com/AladdinDAO/aladdin-v3-contracts
  - initial review commit *c7751c0*
  - final review commit *7edd109*

2.2 Contract List

The following content shows the contracts included in the Concentrator, which the SECBIT team audits:

Name	Lines	Description
AladdinCRV.sol	259	The aCRV token issuance contract represents the share of cvxCRV tokens deposited in the contract.
AladdinConvexVault.sol	383	A core contract where users deposit LP tokens and receive their earnings.
AladdinCRVZap.sol	125	A helper contract to `AladdinCRV`, responsible for the swap of specified tokens.
AladdinConvexVaultZap.sol	204	A helper contract to `AladdinConvexVault`, responsible for the swap of specified tokens.

3. Contract Analysis

This part describes code assessment details, including two items: "role classification" and "functional analysis".

3.1 Role Classification

There are two key roles in the Concentrator: Governance Account and Common Account.

- Governance Account
    - Description

      Contract administrator
    - Authority
        - Update basic parameters
        - Add new Convex pool

        - Transfer ownership
    - Method of Authorization

      The contract administrator is the contract's creator or authorized by the transferring of governance account.

- Common Account
    - Description

      Users participate in the Concentrator.
    - Authority
        - Stake LP token by `AladdinConvexVault`.
        - Retrieval of yield from Convex
        - Convex yield reinvestment
    - Method of Authorization

      No authorization required

3.2 Functional Analysis

The Concentrator allows Convex liquidity providers to stake their LP tokens and get diversified benefits. The SECBIT team conducted a detailed audit of some of the contracts in the protocol. We can divide the critical functions of the contract into two parts:

AladdinCRV

This contract uses the aCRV token to record the cvxCRV tokens deposited into the contract, which will be deposited into Convex for revenue.

The main functions in `AladdinCRV` are as below:

- `deposit()`

  This function allows the user to deposit cvxCRV tokens stored directly into the Convex protocol. At the same time, the recipient specified by the user will receive the corresponding share of aCRV tokens.
- `depositWithCRV()`

The user can call this function to deposit a specified number of CRV tokens into the contract, convert them into cvxCRV tokens, and deposit them in the Convex protocol.

- `withdraw()`

  The user calls this function to retrieve the deposited cvxCRV token, which also requires the aCRV token to be burned in the appropriate amount.

- `harvest()`

  It allows anyone to call this function to retrieve the proceeds of this contract in the Convex protocol. These proceeds will be converted into cvxCRV tokens and deposited again in the Convex protocol.

AladdinConvexVault

This contract supports users to deposit LP tokens which will be deposited under the Convex protocol. The user will be rewarded with aCRV tokens based on the amount of LP tokens deposited.

The main functions in `AladdinConvexVault` are as below:

- `deposit()`

  The user calls this function to deposit the lp token.

- `withdrawAndClaim()`

  Users can withdraw some tokens from a specific pool and claim pending rewards.

- `claim()`

  It allows users to claim pending rewards from a specific pool.

- `harvest()`

  Anyone can call this function to harvest the pending reward and convert it to aCRV token.

4. Audit Detail

This part describes the process, and the detailed results of the audit also demonstrate the problems and potential risks.

4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project from code bug, logical implementation, and potential risks. The process consists of four steps:

- Fully analysis of contract code line by line.
- Evaluation of vulnerabilities and potential risks revealed in the contract code.
- Communication on assessment and confirmation.
- Audit report writing.

4.2 Audit Result

After scanning with adelaide, sf-checker, and badmsg.sender (internal version) developed by SECBIT Labs and open source tools including Mythril, Slither, SmartCheck, and Securify, the auditing team performed a manual assessment. The team inspected the contract line by line, and the result could be categorized into the following types:

Number	Classification	Result
1	Normal functioning of features defined by the contract	✓
2	No obvious bug (e.g., overflow, underflow)	✓
3	Pass Solidity compiler check with no potential error	✓

4	Pass common tools check with no obvious vulnerability	✓
5	No obvious gas-consuming operation	✓
6	Meet with ERC20 standard	✓
7	No risk in low-level call (call, delegatecall, callcode) and in-line assembly	✓
8	No deprecated or outdated usage	✓
9	Explicit implementation, visibility, variable type, and Solidity version number	✓
10	No redundant code	✓
11	No potential risk manipulated by timestamp and network environment	✓
12	Explicit business logic	✓
13	Implementation consistent with annotation and other info	✓
14	No hidden code about any logic that is not mentioned in design	✓
15	No ambiguous logic	✓
16	No risk threatening the developing team	✓
17	No risk threatening exchanges, wallets, and DApps	✓
18	No risk threatening token holders	✓
19	No privilege on managing others' balances	✓
20	No non-essential minting method	✓

21	Correct managing hierarchy	✓

4.3 Issues

4.3.1 The parameter `UserInfo.shares` is not recorded and updated, which directly results in the user not being able to retrieve the principal and rewards.

Risk Type	Risk Level	Impact	Status
Implementation	Medium	Functional failure	Fixed

Description

The parameter `UserInfo.shares` indicates the share of `LP token` deposited by the user in this contract. It is used to calculate the principal deposited by the user and the amount of rewards. The current code uses the parameter `_pool.totalShare` to record the total `shares` deposited into the pool by all users under the current contract. Still, the code for recording the share for a single user is missing. It directly prevents users from receiving their principal and earnings.

```
struct UserInfo {
    // The amount of shares the user deposited.
    uint128 shares;
    // The amount of current accrued rewards.
    uint128 rewards;
    // The reward per share already paid for the user, with
1e18 precision.
    uint256 rewardPerSharePaid;
  }

function deposit(uint256 _pid, uint256 _amount) public
override nonReentrant returns (uint256 share) {
```

```
    ......

    // 3. deposit
    _approve(_lpToken, BOOSTER, _amount);
    IConvexBooster(BOOSTER).deposit(_pool.convexPoolId,
_amount, true);

    uint256 _totalShare = _pool.totalShare;
    uint256 _totalUnderlying = _pool.totalUnderlying;
    uint256 _shares;
    if (_totalShare == 0) {
      _shares = _amount;
    } else {
      _shares = _amount.mul(_totalShare) / _totalUnderlying;
    }
    //@audit records the total share of lp tokens deposited by
all users
    _pool.totalShare = _toU128(_totalShare.add(_shares));
    _pool.totalUnderlying =
_toU128(_totalUnderlying.add(_amount));

    emit Deposit(_pid, msg.sender, _amount);
    return _shares;
  }

function withdrawAndClaim(
    uint256 _pid,
    uint256 _shares,
    uint256 _minOut,
    ClaimOption _option
  ) public override nonReentrant returns (uint256 withdrawn,
uint256 claimed) {
    require(_shares > 0, "AladdinConvexVault: zero share
withdraw");
    require(_pid < poolInfo.length, "AladdinConvexVault:
invalid pool");

    ......
```

```
    // 2. withdraw lp token
    UserInfo storage _userInfo = userInfo[_pid][msg.sender];

    //@audit share of users not updated
    require(_shares <= _userInfo.shares, "AladdinConvexVault:
shares not enough");

    uint256 _totalShare = _pool.totalShare;
    uint256 _totalUnderlying = _pool.totalUnderlying;
    uint256 _withdrawable = _shares.mul(_totalUnderlying) /
_totalShare;
    {
      // take withdraw fee here
      uint256 _fee =
_withdrawable.mul(_pool.withdrawFeePercentage) /
FEE_DENOMINATOR;
      _withdrawable = _withdrawable - _fee; // never overflow
    }

    _pool.totalShare = _toU128(_totalShare - _shares);
    _pool.totalUnderlying = _toU128(_totalUnderlying -
_withdrawable);

 IConvexBasicRewards(_pool.crvRewards).withdraw(_withdrawable,
false);
    IERC20Upgradeable(_pool.lpToken).safeTransfer(msg.sender,
_withdrawable);
    emit Withdraw(_pid, msg.sender, _shares);

    ......
  }
```

Status

The team fixed this issue in commit 2fd0df7.

4.3.2 If the function `_swapCRVToCvxCRV()` returns an incorrect value, the `harvest()` function may fail to execute.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Low	Design logic	Fixed

Description

The function `_swapCRVToCvxCRV()` converts the CRV token under the `AladdinConvexVault` contract to a cvxCRV token. This function provides both Curve item and Convex item for converting CRV tokens to cvxCRV tokens. The function `_swapCRVToCvxCRV()` calls the `deposit()` function under the CrvDepositor contract when exchanging cvxCRV tokens using the Convex protocol. When `_lock == false`, the caller will be compensated by a percentage of the commission deducted to the other user who locked the position for him. In this case, the caller will receive fewer cvxCRV tokens.

The function `_swapCRVToCvxCRV()` does not consider the fee charged by the `CrvDepositor` contract when processing the amount of cvxCRV tokens swapped using the Convex protocol. It results in the return value of the function `_swapCRVToCvxCRV()` is larger than the actual received value. These exchanged cvxCRV tokens would be transferred to the `AladdinCRV` contract. It will directly result in the `IAladdinCRV(_token).deposit()` function failing to execute due to insufficient funds.

```
//@audit loacted in AladdinConvexVault.sol
function harvest(
    uint256 _pid,
    address _recipient,
    uint256 _minimumOut
  ) external override nonReentrant returns (uint256 harvested)
{
    ......
    if (_amount > 0) {
```

```
        IZap(_zap).zap{ value: _amount }(WETH, _amount, CRV, 0);
    }
    _amount = IERC20Upgradeable(CRV).balanceOf(address(this));

    //@audit swap CRV token to cvxCRV token
    _amount = _swapCRVToCvxCRV(_amount, _minimumOut);

    _token = aladdinCRV; // gas saving
    _approve(CVXCRV, _token, _amount);

    //@audit this function may fail due to an insufficient
funds
    uint256 _rewards =
IAladdinCRV(_token).deposit(address(this), _amount);

    ......
  }

//@audit located in AladdinConvexVault.sol
function _swapCRVToCvxCRV(uint256 _amountIn, uint256 _minOut)
internal returns (uint256) {
    ......

    if (useCurve) {
      _approve(CRV, CURVE_CVXCRV_CRV_POOL, _amountIn);
      _amountOut =
ICurveFactoryPool(CURVE_CVXCRV_CRV_POOL).exchange(0, 1,
_amountIn, 0, address(this));
    } else {
      _approve(CRV, CRV_DEPOSITOR, _amountIn);
      IConvexCRVDepositor(CRV_DEPOSITOR).deposit(_amountIn,
false, address(0));

      //@audit take into account the handling fee,
      // the actual number of cvxCRV tokens redeemed by the
user
      // is less than the number of crv tokens.
      _amountOut = _amountIn;
    }
```

```
        return _amountOut;
    }
```

Status

The team has developed different logic for whether or not the convex protocol charges a fee and fixed this issue in commit 2fd0df7.

4.3.3 The function `_claim()` will duplicate the reward when processing aCRV token rewards.

Risk Type	Risk Level	Impact	Status
Implementation	Medium	Faulty logic	Fixed

Description

When a user chooses to claim with the option `ClaimOption.Claim`, he will receive the corresponding aCRV token directly. However, the code does not exit directly after the `option == ClaimOption.Claim` branch is executed, it will continue with the next `withdraw()` operation. Since the parameter `_withdrawOption` is only initialized, the enumeration type will default to `WithdrawOption.Withdraw` at this point, and the caller will also be rewarded with a cvxCRV token. It results in the caller being repeatedly rewarded, which affects the other users' benefit.

```
//@audit located in IAladdinCRV.sol
enum WithdrawOption {
    Withdraw,
    WithdrawAndStake,
    WithdrawAsCRV,
    WithdrawAsCVX,
    WithdrawAsETH
  }

//@audit located in AladdinConvexVault.sol
```

```
function _claim(
    uint256 _amount,
    uint256 _minOut,
    ClaimOption _option
  ) internal returns (uint256) {
    if (_amount == 0) return _amount;

    IAladdinCRV.WithdrawOption _withdrawOption;

    if (_option == ClaimOption.Claim) {
      require(_amount >= _minOut, "AladdinConvexVault:
insufficient output");
      IERC20Upgradeable(aladdinCRV).safeTransfer(msg.sender,
_amount);
    } else if (_option == ClaimOption.ClaimAsCvxCRV) {

      ......
    }

    return IAladdinCRV(aladdinCRV).withdraw(msg.sender,
_amount, _minOut, _withdrawOption);
  }

//@audit located in AladdinCRV.sol
function withdraw(
    address _recipient,
    uint256 _shares,
    uint256 _minimumOut,
    WithdrawOption _option
  ) public override nonReentrant returns (uint256 withdrawn) {
    uint256 _withdrawed = _withdraw(_shares);

    //@audit this branch will be executed
    if (_option == WithdrawOption.Withdraw) {
      require(_withdrawed >= _minimumOut, "AladdinCRV:
insufficient output");
      IERC20Upgradeable(CVXCRV).safeTransfer(_recipient,
_withdrawed);
    } else {
```

```
      ......
    }

    emit Withdraw(msg.sender, _recipient, _shares, _option);
    return _withdrawed;
  }
```

Suggestion

Add the missing `return()` statement directly after retrieving the aCRV token reward to terminate this function.

```
function _claim(
    uint256 _amount,
    uint256 _minOut,
    ClaimOption _option
  ) internal returns (uint256) {
    if (_amount == 0) return _amount;

    IAladdinCRV.WithdrawOption _withdrawOption;
    if (_option == ClaimOption.Claim) {
      require(_amount >= _minOut, "AladdinConvexVault:
insufficient output");
      IERC20Upgradeable(aladdinCRV).safeTransfer(msg.sender,
_amount);

      //@audit add the followed code
      return _amount;
    } else if (_option == ClaimOption.ClaimAsCvxCRV) {

      ......
  }
```

Status

The development team has fixed this issue in commit 2fd0df7.

4.3.4 It may fail for the user to call the `depositWithCRV()` function to deposit the CRV token.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Low	Design logic	Fixed

Description

The function `depositWithCRV()` converts the CRV token deposited by the user into a cvxCRV token stored in the convex protocol. When converting CRV tokens to cvxCRV tokens using the `_zapToken()` function, both Curve and Convex routes are provided. When converting cvxCRV tokens using the Convex route, as discussed in issue 4.3.2, the number of cvxCRV tokens obtained may be less than the number of CRV tokens transferred. In this case, the function `_deposit()` will fail to execute due to insufficient funds.

```
//@audit located in AladdinCRV.sol
function depositWithCRV(address _recipient, uint256 _amount)
public override nonReentrant returns (uint256 share) {
    uint256 _before =
IERC20Upgradeable(CRV).balanceOf(address(this));
    IERC20Upgradeable(CRV).safeTransferFrom(msg.sender,
address(this), _amount);
    _amount =
IERC20Upgradeable(CRV).balanceOf(address(this)).sub(_before);

    //@audit swap CRV token to cvxCRV token
    _amount = _zapToken(_amount, CRV, _amount, CVXCRV);

    //@audit it may fail
    return _deposit(_recipient, _amount);
  }

//@audit located in AladdinCRV.sol
function _zapToken(
```

```
    uint256 _amount,
    address _fromToken,
    uint256 _minimumOut,
    address _toToken
  ) internal returns (uint256) {
    // @audit call AladdinCRVZap.zap() function
    (bool success, bytes memory data) = zap.delegatecall(

 abi.encodeWithSignature("zap(address,uint256,address,uint256)
", _fromToken, _amount, _toToken, _minimumOut)
    );
    require(success, "AladdinCRV: zap failed");
    return abi.decode(data, (uint256));
  }

//@audit located in AladdinCRVZap.sol
function zap(
    address _fromToken,
    uint256 _amountIn,
    address _toToken,
    uint256 _minOut
  ) external payable override returns (uint256) {
    if (_fromToken == THREE_CRV && _toToken == address(0)) {
      ......
    } else if (_fromToken == CRV && _toToken == CVXCRV) {
      // CRV => CVXCRV
      return _swapCRVToCvxCRV(_amountIn, _minOut);
    } else {
      revert("AladdinCRVZap: token pair not supported");
    }
  }

//@audit located in AladdinCRVZap.sol
function _swapCRVToCvxCRV(uint256 _amountIn, uint256 _minOut)
internal returns (uint256) {
    ......

    if (useCurve) {
      .....
```

```
    } else {
      _approve(CRV, CRV_DEPOSITOR, _amountIn);
      IConvexCRVDepositor(CRV_DEPOSITOR).deposit(_amountIn,
false, address(0));
      _amountOut = _amountIn;
    }
    return _amountOut;
  }
```

Status

The team has developed different logic for whether or not the convex protocol charges a fee and fixed this issue in commit 2fd0df7.

4.3.5 Potential loss of profits in `aladdinConvexVault` contract.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Description

A front-running/MEV bot could first deposit a large amount of LP token into the contract via the `deposit()` function, then execute `harvest()` to update the rewards, and finally execute `withdrawAndClaim()` to take the principal and most of the new rewards. If the rewards are large enough, the attacker may make a profit with the help of a flash loan.

Status

This issue has been discussed. As long as a valid incentive exists for the caller invoking harvest, then the rewards to be distributed will be harvested before it is worthwhile to implement a front-running attack. Considering the reward rate of pools on Convex and the cost of implementing an attack, we believe that this risk is low in the current case.

5. Conclusion

After auditing and analyzing the Concentrator contract, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above.

Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability, and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company, or investment.

APPENDIX

Vulnerability/Risk Level Classification

Level	Description
High	Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock assets inside the contract.
Medium	Damage contract's security under given conditions and cause impairment of benefit for stakeholders.
Low	Cause no actual impairment to contract.
Info	Relevant to practice or rationality of the smart contract, could possibly bring risks.

SECBIT Lab is devoted to constructing a common-consensus, reliable, and ordered blockchain economic entity.

https://secbit.io

audit@secbit.io

@secbit_io