Security Audit Report

AladdinCVX (abcCVX) by AladdinDAO

SECBIT

February 3, 2023

1. Introduction

The AladdinDAO is a decentralized network to shift crypto investments from venture capitalists to the wisdom of crowds through collective value discovery. The AladdinCVX (abcCVX) provides an AMO mechanism that adjusts the ratio of LP tokens to clevCVX tokens under the abcCVX contract, allowing users holding abcCVX tokens to gain greater returns. SECBIT Labs conducted an audit from November 27 to December 9, 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).

Updated on February 2, 2023. The new contract adds a `strategyBalance()` function; modifies the `harvest()` function to include a platform fee; and adjusts the precision of the ratio to suit real scenarios. SECBIT Labs has audited these changes from January 24 to February 2, 2023, and confirmed that they do not affect the security of the contract.

Type	Description	Level	Status
Design & Implementation	4.3.1 Improper initialization used for upgradable contracts.	Info	Discussed
Design & Implementation	4.3.2 The implementation logic of the function `addLiquidity()` is inconsistent with the comment.	Info	Discussed
Gas Optimization	4.3.3 Unused parameters.	Info	Fixed
Design & Implementation	4.3.4 The `unlock()` function uses the `_debtBalanceInContract()` function incorrectly when calculating the `_lpBalance` parameter.	High	Fixed
Design & Implementation	4.3.5 Discussion of the return value of `_harvest()` function.	Info	Discussed
Design & Implementation	4.3.6 Discussion of the abcCVX token mint strategy.	Info	Discussed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the AladdinCVX (abcCVX) Protocol is shown below:

- Smart contract code
    - initial review commit *c8f14d0*
    - final review commit *7c20a51*
- *Updated on February 2, 2023.* AladdinCVX (abcCVX) update.
    - review commit e270925

2.2 Contract List

The following content shows the contracts included in the AladdinCVX (abcCVX) Protocol, which the SECBIT team audits:

Name	Lines	Description
AladdinCVX.sol	236	The core contract to rebalance the Curve pool based on tokens in the pool.
AMOMath.sol	115	Auxiliary library contract to calculate Curve liquidity and swap data.
CLeverAMOBase.sol	292	The `baseToken` deposited by the user will be converted into shares to receive the proceeds. The user can withdraw the shares at any time, at which point the user gets the `baseToken` back.
RewardClaimable.sol	61	Holders of abcCVX tokens can harvest their rewards through this contract.
AMOConvexCurveStrategy.sol	10	Proxy contracts, interfaced with the Convex protocol, deposit user funds under the Convex protocol and receive the reward.

Updated on February 2, 2023. The `CLeverAMOBase.sol` *contract has changed the value of the base ratio from* `1e18` *to* `1e10`*. The* `CLeverAMOBase.sol` *contract adds the option to charge a platform fee. In addition, a new* `AMOConvexCurveStrategy.sol` *contract has been introduced.*

3. Contract Analysis

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

3.1 Role Classification

Two key roles in the AladdinCVX (abcCVX) Protocol are Governance Account and Common Account.

- Governance Account
    - Description

      Contract Administrator
    - Authority
        - Update basic parameters
        - Rebalance Curve pool
        - Transfer ownership
    - Method of Authorization

      The contract administrator is the contract's creator or authorized by transferring the governance account.
- Common Account
    - Description

      Users participate in the AladdinCVX (abcCVX) Protocol.
    - Authority
        - Deposit `baseToken` to receive a share token
        - Harvest pending rewards and reinvest in the pool
    - Method of Authorization

      No authorization required

3.2 Functional Analysis

Users can get abcCVX tokens by depositing CVX tokens into the AladdinCVX (abcCVX) Protocol. Through the AMO mechanism, they can earn higher reinvestment returns. 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 three parts:

AladdinCVX, CLeverAMOBase, and RewardClaimable

These three contracts cover the core functionality of the AladdinCVX (abcCVX) Protocol. The main functions in these contracts are as below:

- `rebalance()`

  This function allows the contract owner to rebalance the curve pool based on the token balance ratio in the Curve pool.

- `deposit()`

  Users can deposit base tokens to the contract, which will be locked for a specific period.

- `unlock()`

  The user calls this function to unlock the base tokens (CVX tokens). The funds are split into two: to add liquidity to the Curve pool and deposit the obtained lp tokens into the corresponding gauge and to exchange into clevCVX tokens and deposit them under the Aladdin Furnace protocol. Ultimately, the user receives a share of tokens.

- `withdraw()`

  It will burn share tokens (abcCVX token), and the user can withdraw to debt token and lp token according to the current ratio.

- `withdrawToBase()`

  It will burn shares and withdraw to base tokens (CVX tokens).

- `harvest()`

  Any user could harvest the pending rewards and reinvest them in the pool.

- `claim()`

  Those who hold abcCVX tokens claim pending rewards from the contract.

4. Audit Detail

This part describes the process, and the detailed audit results 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 bugs, 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 the 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 Improper initialization used for upgradable contracts.

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

Description

Due to a requirement of the proxy-based upgradeability system, no constructor can be used in upgradeable contracts. The code within an implementation contract's constructor will never be executed in the context of the proxy's state.

When writing an upgradable contract, you need to change its constructor into a regular function, typically named `initialize`, where you run all the setup logic. To prevent a contract from being *initialized* multiple times, you must add a check to ensure the `initialize` function is called only once. For more details, see the link: https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable.

The same issue exists in the `CLeverAMOBase.sol` contract.

```
constructor(
    address _baseToken,
    address _debtToken,
    address _curvePool,
    address _curveLpToken,
    address _furnace,
    address _gauge,
    address _minter
  ) CLeverAMOBase(_baseToken, _debtToken, _curvePool,
_curveLpToken, _furnace) {
    gauge = _gauge;
    minter = _minter;

    address _coin0 = ICurveFactoryPlainPool(_curvePool).coins(0);
    debtIndex = _coin0 == _baseToken ? 1 : 0;
    baseIndex = _coin0 == _baseToken ? 0 : 1;
  }
```

```
//@audit located in CLeverAMOBase.sol
constructor(
    address _baseToken,
    address _debtToken,
    address _curvePool,
    address _curveLpToken,
    address _furnace
  ) {
    baseToken = _baseToken;
    debtToken = _debtToken;
    curvePool = _curvePool;
    curveLpToken = _curveLpToken;
    furnace = _furnace;
  }
```

Status

Since the constructors in related contracts only assign values to immutable type variables and do not affect contract storage, there is no need to modify it here.

4.3.2 The implementation logic of the function `addLiquidity()` is inconsistent with the comment.

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

Description

According to the explanation of the parameter `supply` in the `addLiquidity()` function, it represents the total number of lp tokens in the Curve pool, but the `supply` calculated in the current code finally means the number of new lp tokens minted, not the total number of lp tokens. Also, in the earlier version of the code (removed now), the `add_out` parameter is commented as `The current total supply of curve pool` when using the `addLiquidity()` function. According to the comment, the `add_out` parameter means the total number of lp tokens, not the number of new lp tokens. Here there is a contradiction between the comment and the code implementation.

```
/// @param amp The amplification parameter equals: A n^(n-1)
/// @param fee The swap fee from curve pool.
/// @param supply The current total supply of curve pool
/// @param x The balance of token0.
/// @param y The balance of token1.
/// @param dx The input amount of token0.
/// @param dy The input amount of token1.
function addLiquidity(
    uint256 amp,
    uint256 fee,
    uint256 supply,
    uint256 x,
    uint256 y,
    uint256 dx,
    uint256 dy
  )
    internal
    pure
    returns (
      uint256,
      uint256,
      uint256
    )
  {
    fee = fee / 2; // the base_fee for each token
    uint256 invariant0 = getInvariant(amp, x, y);

    .......
    // compute new minted
    dx -= diff_x;
    dy -= diff_y;
    invariant1 = getInvariant(amp, dx, dy);

    //@audit new lp tokens
    supply = (supply * (invariant1 - invariant0)) / invariant0;

    return (x, y, supply);
  }
```

Suggestion

As the function reuses the variables in the incoming parameters as return values, it is prone to ambiguity. The actual meaning of the relevant variables needs to be clarified.

Status

The team clarified this parameter's meaning and modified the comment in commit 7c20a51.

4.3.3 Unused parameters.

Risk Type	Risk Level	Impact	Status
Gas Optimization	Info	More gas consumption	Fixed

Description

The parameter `MAX_PLATFORM_FEE` is never used. If there are no special considerations, this parameter can be removed to reduce gas consumption during contract deployment.

```
/// @dev The maximum value of platform fee percentage.
uint256 private constant MAX_PLATFORM_FEE = 2e8; // 20%
```

Status

The team removes this code in commit 7c20a51.

4.3.4 The `unlock()` function uses the `_debtBalanceInContract()` function incorrectly when calculating the `_lpBalance` parameter.

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

Description

The _lpBalance parameter indicates the number of lp tokens deposited under the Curve Protocol for this contract. However, the current calculation of the _lpBalance parameter uses the _debtBalanceInContract() function incorrectly, which represents the number of clevCVX tokens that have not yet been exchanged into CVX tokens under the Aladdin Furnace Protocol for this contract.

```
function unlock(uint256 _minShareOut) external override returns
(uint256 _shares) {
    ......
    uint256 _totalSupply = totalSupply();
    if (_totalSupply == 0) {
      // choose max(_debtOut, _lpOut) as initial supply
      _shares = _debtOut > _lpOut ? _debtOut : _lpOut;
    } else {


      // This already contains the user converted amount, we need
to subtract it when computing shares.
      uint256 _debtBalance = _debtBalanceInContract();

      // @audit misuse
      uint256 _lpBalance = _debtBalanceInContract();

      _debtOut = (_debtOut * _totalSupply) / (_debtBalance -
_debtOut);
      _lpOut = (_lpOut * _totalSupply) / (_lpBalance - _lpOut);

      // use min(debt share, lp share) as new minted sharey
      _shares = _debtOut < _lpOut ? _debtOut : _lpOut;
    }

    require(_shares >= _minShareOut, "CLeverAMO: insufficient
shares");

    _mint(msg.sender, _shares);

    emit Unlock(msg.sender, _unlocked, _shares, ratio());
  }
```

Suggestion

Calculate the `_lpBalance` parameter using the `_lpBalanceInContract()` function, modified as follows.

```
function unlock(uint256 _minShareOut) external override returns
(uint256 _shares) {
    ......

    uint256 _totalSupply = totalSupply();
    if (_totalSupply == 0) {
      // choose max(_debtOut, _lpOut) as initial supply
      _shares = _debtOut > _lpOut ? _debtOut : _lpOut;
    } else {


      // This already contains the user converted amount, we need
to subtract it when computing shares.
      uint256 _debtBalance = _debtBalanceInContract();

      // @audit use _lpBalanceInContract function instead
      //        of _debtBalanceInContract
      uint256 _lpBalance = _lpBalanceInContract();

      _debtOut = (_debtOut * _totalSupply) / (_debtBalance -
_debtOut);
      _lpOut = (_lpOut * _totalSupply) / (_lpBalance - _lpOut);

      // use min(debt share, lp share) as new minted sharey
      _shares = _debtOut < _lpOut ? _debtOut : _lpOut;
    }

    require(_shares >= _minShareOut, "CLeverAMO: insufficient
shares");

    _mint(msg.sender, _shares);

    emit Unlock(msg.sender, _unlocked, _shares, ratio());
  }
```

Status

The team has adopted the suggestion and fixed this issue in commit 7c20a51.

4.3.5 Discussion of the return value of `_harvest()` function.

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

Description

According to the current logic, the `_harvest()` function withdraws an extra base tokens (CVX) reward. On the one hand, the return value of the internal function `_harvest()` is always zero, and there seems to be no need for the accumulation in the `harvest()` function. On the other hand, the rewards currently claimed in the `_harvest()` function do not contain base tokens (CVX) either. Therefore, it is essential to check if the implementation is complete here.

```
//@audit located in CLeverAMOBase.sol
function harvest(address _recipient, uint256 _minBaseOut)
external override returns (uint256 _baseTokenOut) {
    // claim from furnace
    _baseTokenOut = _claimBaseFromFurnace();

    // @audit the _harvest() function always returns zero.
    // harvest external rewards
    _baseTokenOut += _harvest();

    require(_baseTokenOut >= _minBaseOut, "CLeverAMO:
insufficient harvested");

    uint256 _bounty = (_baseTokenOut * bountyPercentage) /
FEE_PRECISION;

    (uint256 _debtAmount, uint256 _lpAmount, uint256 _ratio) =
_convertFromBaseToken(_baseTokenOut - _bounty);
    _depositDebtToken(_debtAmount);
    _depositLpToken(_lpAmount);
    emit Harvest(_recipient, _baseTokenOut, _bounty, _debtAmount,
_lpAmount, _ratio);

    if (_bounty > 0) {
      IERC20Upgradeable(baseToken).safeTransfer(_recipient,
_bounty);
    }
  }

//@audit located in AladdinCVX.sol
function _harvest() internal override returns (uint256) {
    uint256 _length = rewards.length;
    uint256[] memory _amounts = new uint256[](_length);
    for (uint256 i = 0; i < _length; i++) {
      address _token = rewards[i];
      _amounts[i] =
IERC20Upgradeable(_token).balanceOf(address(this));
    }
    // claim CRV  ()
    ICurveMinter(minter).mint(gauge);
    // claim extra rewards
    ICurveGauge(gauge).claim_rewards();

    uint256 _totalSupply = totalSupply();
    for (uint256 i = 0; i < _length; i++) {
      address _token = rewards[i];
      _amounts[i] =
IERC20Upgradeable(_token).balanceOf(address(this)) - _amounts[i];

      rewardPerShare[_token] += (_amounts[i] * REWARD_PRECISION)
/ _totalSupply;
    }

    return 0;
  }
```

Status

The team confirmed this logic and decided to maintain the current design.

4.3.6 Discussion of the abcCVX token mint strategy.

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

Description

The base tokens (CVX) deposited into the contract will be split into two parts after unlocking: a portion of the base tokens will be used to add liquidity, and the resulting lp token will be deposited into the gauge. The remaining base tokens (CVX) will be exchanged into clevCVX tokens via the Curve Protocol and deposited into the Aladdin Furnace. Then, a share will be calculated based on the number of base tokens in each section, and the smaller of these will be the number of abcCVX tokens received by the user. This design logic may result in the user getting fewer funds back when they burn the abcCVX token.

```
function unlock(uint256 _minShareOut) external override returns
(uint256 _shares) {
    ......
    // convert to debt token and lp token
    (uint256 _total, uint256 _debtOut, uint256 _lpOut) =
_checkpoint(_unlocked);
    _debtOut = (_debtOut * _unlocked) / _total;
    _lpOut = (_lpOut * _unlocked) / _total;

    uint256 _totalSupply = totalSupply();
    if (_totalSupply == 0) {
      // choose max(_debtOut, _lpOut) as initial supply
      _shares = _debtOut > _lpOut ? _debtOut : _lpOut;
    } else {
      // This already contains the user converted amount, we need
to subtract it when computing shares.
      uint256 _debtBalance = _debtBalanceInContract();
      uint256 _lpBalance = _debtBalanceInContract();
      _debtOut = (_debtOut * _totalSupply) / (_debtBalance -
_debtOut);
      _lpOut = (_lpOut * _totalSupply) / (_lpBalance - _lpOut);

      // use min(debt share, lp share) as new minted sharey
      _shares = _debtOut < _lpOut ? _debtOut : _lpOut;
    }

    require(_shares >= _minShareOut, "CLeverAMO: insufficient
shares");

    _mint(msg.sender, _shares);

    emit Unlock(msg.sender, _unlocked, _shares, ratio());
  }
```

Status

This issue has been discussed. The team confirmed the design logic. The scenarios we construct will not happen in practice.

5. Conclusion

After auditing and analyzing the AladdinCVX (abcCVX) Protocol 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