# Security Audit Report

## CLever by AladdinDAO

SECBIT

April 21, 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. The CLever protocol helps users automatically lock CVX tokens into the Convex protocol. It will automatically retrieve the rewards and convert them into CVX tokens. It also provides the lending of clevCVX tokens, which can be converted into CVX tokens further. SECBIT Labs conducted an audit from April 11 to April 21, 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 |
|---|---|---|---|
| Design & Implementation | 4.3.1 Simplify the fetching of `_index` to reduce the complexity of the calculation. | Info | Fixed |
| Implementation | 4.3.2 Discussion of `processUnlockableCVX()` function. | Info | Discussed |
| Implementation | 4.3.3 Discussion of Parameter `_burnAmount`. | Info | Fixed |
| Design & Implementation | 4.3.4 Discussion of `harvest()` function. | Low | Fixed |
| Design & Implementation | 4.3.5 Others can claim votium rewards for the protocol. | Low | Fixed |
| Design & Implementation | 4.3.6 Potential DOS risks in Vesting contract. | Low | Fixed |

# 2. Contract Information

This part describes the basic contract information and code structure.

## 2.1 Basic Information

The basic information about the CLever contract is shown below:

- Project website
    - https://clever.aladdin.club/
- Smart contract code
    - initial review commit *ec40674*
    - final review commit *2822b56*

## 2.2 Contract List

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

| Name | Lines | Description |
|---|---|---|
| CLeverToken.sol | 54 | A CLever Token issuance contract that inherits the standard ERC20 token. |
| CLeverCVXLocker.sol | 580 | This contract helps users automatically lock CVX tokens and provide clevCVX token lending services. |
| Furnace.sol | 315 | This contract offers a service to exchange clevCVX tokens for CVX tokens. |
| Vesting.sol | 61 | A token vesting contract. |

*Other contracts in `governance` directory are forked from Curve DAO with minor parameter changes.*

# 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 CLever: Governance Account and Common Account.

- Governance Account
  - Description

    Contract administrator
  - Authority
    - Update basic parameters
    - Update the percentage of various fees charged
    - Transfer ownership
  - Method of Authorization

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

    Users participate in the Aladdin CLever.
  - Authority
    - Deposit CVX tokens and receive clevCVX tokens reward
    - Borrow clevCVX token
    - Exchange clevCVX tokens for CVX tokens
  - Method of Authorization

    No authorization required

## 3.2 Functional Analysis

The CLever protocol automatically locks the CVX token deposited by the user into the Convex protocol. It also provides the user with a leveraged lending service to significantly increase the user's revenue. 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:

### CLeverCVXLocker

Users can automatically utilize this contract to lock CVX tokens into the Convex protocol. They will also be able to use the leveraged lending service on top of the clevCVX tokens reward.

The main functions in `CLeverCVXLocker` are as below:

- `deposit()`

  This function allows the user to deposit CVX tokens into this contract, which will be locked into the Convex protocol.

- `unlock()`

  The user calls this function to unlock their deposited CVX token, which must wait for 17 epochs before it can be retrieved.

- `withdrawUnlocked()`

  The user calls this function to retrieve CVX tokens that satisfy conditions.

- `repay()`

  When users use the lending service, they can call on this function to repay their debt. They can choose to repay the debt in the form of CVX tokens or clevCVX tokens.

- `borrow()`

  This function provides clevCVX tokens lending service.

- `harvest()`

  Any user can call this function to retrieve rewards of this contract in the Convex protocol. These rewards will be converted into CVX tokens, which will be distributed to depositors.

- `processUnlockableCVX()`

  Any user can call this function to process an unlocked CVX token in a contract. The extra CVX tokens in the contract will be relocked into the Convex protocol by default.

### Furnace

This contract provides a service to exchange clevCVX tokens for CVX tokens. If there are insufficient CVX tokens under this contract, the user will have to wait. The main functions in `Furnace` are as below:

- `deposit()`

  Deposit clevCVX token in this contract to change for CVX token.

- `withdraw()`

  The user retrieves the unexchanged clevCVX tokens to the specified address.

- `claim()`

  The user retrieves the exchanged CVX tokens and burns the corresponding amount of clevCVX tokens.

- `harvest()`

  This function retrieves the contract's earnings and converts them into CVX tokens.

# 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 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 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 Simplify the fetching of `_index` to reduce the complexity of the calculation.

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Implementation | Info | Code revise | Fixed |

**Description**

The getUserLocks() function displays the amount of CVX tokens that the specified user can unlock and is waiting to unlock. It uses the epochLocked[17] array to keep track of the number of CVX tokens deposited at different epochs. The formula (_currentEpoch + i + 1) % 17 is used to calculate the corresponding _index when querying whether a given user has deposited CVX tokens under different arrays. Since the loop only needs to determine the number of non-empty arrays, regardless of when the user actually deposited them, traversing the _index parameter directly from 0 to 16 gives a more straightforward result and reduces the size of the calculation.

```
//@audit located in CLeverCVXLocker.sol
function getUserLocks(address _account)
    external
    view
    returns (EpochUnlockInfo[] memory locks, EpochUnlockInfo[] memory
pendingUnlocks)
  {
    UserInfo storage _info = userInfo[_account];

    uint256 _currentEpoch = block.timestamp / REWARDS_DURATION;
    uint256 lengthLocks;
    for (uint256 i = 0; i < 17; i++) {
      //@audit modify index fetch method
      uint256 _index = (_currentEpoch + i + 1) % 17;
      if (_info.epochLocked[_index] > 0) {
        lengthLocks++;
      }
    }

    locks = new EpochUnlockInfo[](lengthLocks);
    lengthLocks = 0;
    for (uint256 i = 0; i < 17; i++) {
      uint256 _index = (_currentEpoch + i + 1) % 17;
      if (_info.epochLocked[_index] > 0) {
        locks[lengthLocks].pendingUnlock =
uint192(_info.epochLocked[_index]);
        locks[lengthLocks].unlockEpoch = uint64(_currentEpoch + i + 1);
        lengthLocks += 1;
      }
    }
    ......
  }
```

### Suggestion

The corresponding simplified scheme is as follows.

```
//@audit located in CLeverCVXLocker.sol
function getUserLocks(address _account)
    external
    view
    returns (EpochUnlockInfo[] memory locks, EpochUnlockInfo[] memory
pendingUnlocks)
  {
    UserInfo storage _info = userInfo[_account];

    uint256 _currentEpoch = block.timestamp / REWARDS_DURATION;
    uint256 lengthLocks;
    for (uint256 i = 0; i < 17; i++) {
       // @audit delete the following code
      //uint256 _index = (_currentEpoch + i + 1) % 17;
      if (_info.epochLocked[i] > 0) {
        lengthLocks++;
      }
    }

    locks = new EpochUnlockInfo[](lengthLocks);
    lengthLocks = 0;
    for (uint256 i = 0; i < 17; i++) {
      uint256 _index = (_currentEpoch + i + 1) % 17;
      if (_info.epochLocked[_index] > 0) {
        locks[lengthLocks].pendingUnlock =
uint192(_info.epochLocked[_index]);
        locks[lengthLocks].unlockEpoch = uint64(_currentEpoch + i + 1);
        lengthLocks += 1;
      }
    }
    ......
  }
```

### Status

The team fixed this issue in commit 000d36a.

### 4.3.2 Discussion of `processUnlockableCVX()` function.

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Implementation | Info | Design logic | Discussed |

**Description**

The function `processUnlockableCVX()` is used to retrieve unlocked CVX tokens from the Convex protocol, including CVX tokens that the user is ready to unlock. Therefore, to ensure that users in the CLever protocol can retrieve their CVX token funds that were ready to be unlocked 17 (possibly earlier) epochs ago, it is recommended that someone proactively call the `processUnlockableCVX()` function once per epoch after the protocol has officially run for 17 epochs, to prevent the occurrence of `pendingUnlocked[currentEpoch]` is not processed. If the current epoch is not called, then its corresponding `pendingUnlocked` may never be processed.

```
//@audit located in CLeverCVXLocker.sol
function processUnlockableCVX() external {
    // Be careful that someone may kick us out from CVXLockerV2
    // `totalUnlockedGlobal` keep track the amount of CVXunlocked from
CVXLockerV2
    // all other CVXin this contract can be considered unlocked from
CVXLockerV2 by someone else.

    // 1. find extra CVXfrom donation or kicked out from CVXLockerV2
    uint256 _extraCVX= totalCVXInPool().sub(totalUnlockedGlobal);

    // 2. unlock CVX
    uint256 _unlocked = IERC20Upgradeable(CVX).balanceOf(address(this));
    IConvexCVXLocker(CVX_LOCKER).processExpiredLocks(false);
    _unlocked =
IERC20Upgradeable(CVX).balanceOf(address(this)).sub(_unlocked).add(_extra
CVX);

    // 3. remove user unlocked CVX
    uint256 currentEpoch = block.timestamp / REWARDS_DURATION;
    //@audit Only the number of CVXs that need to be unlocked for the
current epoch can be processed
    uint256 _pending = pendingUnlocked[currentEpoch];
    if (_pending > 0) {
      // check if the unlocked CVXis enough, normally this should always
be true.
      require(_unlocked >= _pending, "CLeverCVXLocker: insufficient
unlocked CVX");
      _unlocked -= _pending;
      // update global info
      totalUnlockedGlobal = totalUnlockedGlobal.add(_pending);
      totalPendingUnlockGlobal -= _pending; // should never overflow
      pendingUnlocked[currentEpoch] = 0;
    }

    // 4. relock
    if (_unlocked > 0) {
      IERC20Upgradeable(CVX).safeApprove(CVX_LOCKER, 0);
      IERC20Upgradeable(CVX).safeApprove(CVX_LOCKER, _unlocked);
      IConvexCVXLocker(CVX_LOCKER).lock(address(this), _unlocked, 0);
    }
  }
```

**Status**

This issue has been discussed. The team will ensure this function is called once per epoch.

### 4.3.3 Discussion of Parameter `_burnAmount`.

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Implementation | Info | Meaningless variable | Fixed |

**Description**

The internal function `_distribute()` is used to repay the clevCVX token credited to the protocol. The contract will execute the `_distribute()` function to process the clevCVX token to be exchanged in the `Transmuter` contract when the user repays the CVX token to the project or when this protocol generates CVX tokens proceeds. The parameter `_burnAmount` records the actual number of clevCVX tokens to be exchanged when the `_distribute()` function is executed. This parameter is not actually used in the contract, and its use needs to be clarified here.

```
//@audit located in Transmuter.sol (Furnace.sol)
function _distribute(address _origin, uint256 _amount) internal {
    distributeIndex += 1;
    uint256 _totalUnrealised = totalUnrealised;
    uint256 _totalRealised = totalRealised;
    uint128 _accUnrealisedFraction = accUnrealisedFraction;
    uint256 _burnAmount;
    // 1. distribute CVX rewards
    if (_amount >= _totalUnrealised) {
      // In this case, all unrealised clevCVX are paid off.
      totalUnrealised = 0;
      totalRealised = _toU128(_totalUnrealised + _totalRealised);

      accUnrealisedFraction = 0;
      lastPaidOffDistributeIndex = distributeIndex;

      //@audit variable not actually used
      _burnAmount = _totalUnrealised;
    } else {
      totalUnrealised = uint128(_totalUnrealised - _amount);
      totalRealised = _toU128(_totalRealised + _amount);

      uint128 _fraction = _toU128(((_totalUnrealised - _amount) * E128) /
_totalUnrealised); // mul never overflow
      accUnrealisedFraction = _mul128(_accUnrealisedFraction, _fraction);

      //@audit variable not actually used
      _burnAmount = _amount;
    }

    // 2. stake extra CVX to CVXRewardPool
    uint256 _toStake =
totalCVXInPool().mul(stakePercentage).div(FEE_DENOMINATOR);
    uint256 _balanceStaked =
IConvexCVXRewardPool(CVX_REWARD_POOL).balanceOf(address(this));
    if (_balanceStaked < _toStake) {
      _toStake = _toStake - _balanceStaked;
      if (_toStake >= stakeThreshold) {
        IERC20Upgradeable(CVX).safeApprove(CVX_REWARD_POOL, 0);
        IERC20Upgradeable(CVX).safeApprove(CVX_REWARD_POOL, _toStake);
        IConvexCVXRewardPool(CVX_REWARD_POOL).stake(_toStake);
      }
    }

    emit Distribute(_origin, _amount);
  }
```

**Status**

The team deletes this variable in commit 000d36a.

### 4.3.4 Discussion of `harvest()` function.

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

**Description**

The parameter `_minimumOut` in the `harvest()` function is designed to prevent risks such as price manipulation during the exchange of cvxcrv tokens into CVX tokens. Instead of requiring the user to transfer funds into the contract, this function processes the proceeds retrieved from this contract directly and converts them into CVX tokens. However, given that `_minimumOut` is passed in by the user and `harvest()` can be called by anyone, the risk of price manipulation still needs to be considered. Specifically, `harvestVotium` may be riskier as its rewards may be distributed instantaneously when the time is up, the funds may be significant, and some asset pools may have poor liquidity, making them riskier to MEV bots.

```
//@audit located in Transmuter.sol (Furnace.sol)
function harvest(address _recipient, uint256 _minimumOut) external
returns (uint256) {
    // 1. harvest from CVXRewardPool
    IConvexCVXRewardPool(CVX_REWARD_POOL).getReward(false);

    // 2. swap all reward to CVX (cvxCRV only currently)
    uint256 _amount = IERC20Upgradeable(CVXCRV).balanceOf(address(this));
    if (_amount > 0) {
      IERC20Upgradeable(CVXCRV).safeTransfer(zap, _amount);
      _amount = IZap(zap).zap(CVXCRV, _amount, CVX, _minimumOut);
    }

    emit Harvest(msg.sender, _amount);

    if (_amount > 0) {
      uint256 _distributeAmount = _amount;
      // 3. take platform fee and harvest bounty
      uint256 _platformFee = platformFeePercentage;
      if (_platformFee > 0) {
        _platformFee = (_platformFee * _distributeAmount) /
FEE_DENOMINATOR;
        IERC20Upgradeable(CVX).safeTransfer(platform, _platformFee);
        _distributeAmount = _distributeAmount - _platformFee; // never
overflow here
      }
    ......
  }

function harvestVotium(IVotiumMultiMerkleStash.claimParam[] calldata
claims, uint256 _minimumOut)
    external
    override
    returns (uint256)
  {
    // 1. claim reward from votium
    IVotiumMultiMerkleStash(VOTIUM_DISTRIBUTOR).claimMulti(address(this),
claims);
    address[] memory _rewardTokens = new address[](claims.length);
    uint256[] memory _amounts = new uint256[](claims.length);
    for (uint256 i = 0; i < claims.length; i++) {
      _rewardTokens[i] = claims[i].token;
      _amounts[i] = claims[i].amount;
    }

  ......
  }
```

**Status**

This issue has been discussed and the team added keeper role to restrict `harvestVotium()` function calls in commit 000d36a.

### 4.3.5 Others can claim votium rewards for the protocol.

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

**Description**

Others can claim rewards from votium for the protocol. The rewarded tokens can be transferred directly into the contract, thus breaking some of the assumptions of the current implementation.

```
  // @audit located in CLeverCVXLocker.sol
  function harvestVotium(IVotiumMultiMerkleStash.claimParam[] calldata
claims, uint256 _minimumOut)
    external
    override
    onlyKeeper
    returns (uint256)
  {
    // 1. claim reward from votium
    IVotiumMultiMerkleStash(VOTIUM_DISTRIBUTOR).claimMulti(address(this),
claims); // @audit what if someone claimed rewards for CLever
    ...
  }
```

**Suggestion**

Consider this case and verify that the current code still conforms to the design.

**Status**

Fixed in commit 2822b56.

### 4.3.6 Potential DOS risks in Vesting contract.

| Risk Type | Risk Level | Impact | Status |
|---|---|---|---|
| Design & Implementation | Low | DOS | Fixed |

**Description**

One can open `newVesting` for anyone, thus making it impossible for the victim to create a new one. It could lead to a potential denial of service attack.

```
    // @audit located in Vesting.sol
    function newVesting(
      address _recipient,
      uint256 _amount,
      uint256 _startTime,
      uint256 _endTime
    ) external {
      require(vesting[_recipient].vestedAmount == 0, "Vesting: already
vested");
      require(_startTime < _endTime && _endTime < uint64(-1), "Vesting:
invalid timestamp");

      IERC20(token).safeTransferFrom(msg.sender, address(this), _amount);
      ...
    }
```

### Suggestion

Change the design of the `newVesting` function.

### Status

Fixed in commit 2822b56.

## 5. Conclusion

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

## Disclaimer

# 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