Security Audit Report

Generic Version of CLever by AladdinDAO

SECBIT

July 1, 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 new CLever protocol adds more application scenarios to the original. It offers three different asset yield strategies. The user can choose each strategy according to their needs. SECBIT Labs conducted an audit from May 19 to July 1, 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 Discussion of debt repayment mechanisms.	Info	Discussed
Design & Implementation	4.3.2 Discussion of the parameter `expectedUnderlyingTokenAmount`.	Info	Discussed
Design & Implementation	4.3.3 Add judgement on the parameter `_share` to optimize the code structure.	Info	Discussed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the CLever extension contract is shown below:

- Project website
  - https://clever.aladdin.club/
- Smart contract code
  - initial review commit *fe9e759*
  - final review commit *53152b1*

2.2 Contract List

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

Name	Lines	Description
MataCLever.sol	646	This contract helps users automatically deposit target tokens and provide clevCVX token lending services.
MetaFurnace.sol	315	This contract offers a service to exchange clevCRV tokens for CRV tokens.
AladdinCRVStrategy.sol	58	Specific strategy for CRV token and aCRV token.
ConcentratorBatchStrategy.sol	53	Concentrator batch strategy for CLever protocol.
ConcentratorStrategy.sol	151	Concentrator strategy for CLever protocol.
YieldStrategyBase.sol	40	Contracts that handle yield 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 CLever extension: 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 protocol.

  - Authority
    - Deposit underlying tokens and receive rewards
    - Exchange clevCRV tokens for CRV tokens
  - Method of Authorization

    No authorization required

3.2 Functional Analysis

The CLever extension protocol automatically locks the underlying token deposited by the user into the Convex protocol. In addition, this protocol provides for exchanging clevCRV tokens and CRV tokens. 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:

MataCLever

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

The main functions in `MataCLever` are as below:

- `deposit()`

This function allows the user to deposit the underlying token or yield token as a credit to this contract.

- `withdraw()`

  This function allows the user to withdraw the underlying token or yield token from this contract.

- `repay()`

  Users can call on this function to repay their debt when they use the lending service.

- `mint()`

  A borrower can mint a certain amount of debt tokens.

- `burn()`

  The caller can burn a certain amount of debt tokens from the caller's balance to pay the debt for someone.

- `claim()`

  Users can call this function to claim extra rewards from a specific strategy.

- `claimAll()`

  Users can call this function to claim extra rewards from all deposited strategies.

- `harvest()`

  Users can call this function to harvest rewards from the corresponding yield strategy.

MetaFurnace

This contract provides a service to exchange clevCRV tokens for CRV tokens. The main functions in `MetaFurnace` are as below:

- `deposit()`

  Deposit clevCRV token in this contract to change for CRV token.

- `withdraw()`

The user withdraws the unrealized debt token of the caller from this contract.

- `claim()`

  The user claims all realized baseToken of the caller from this contract.

- `distribute()`

  The whitelist users distribute base tokens from the origin address to pay the debt.

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 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 Discussion of debt repayment mechanisms.

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

Description

The `MetaCLever` contract provides different yield strategies, each of which has a different type of `underlying token` and `yield token`. In the current code logic, administrators can set their own `underlying token` and `yield token`. Consider the following scenario: Suppose that the `underlying token` is WETH and the `yield token` is `clevcvx token` in a current strategy. UserA calls `deposit()` function to deposit 10 WETH into the contract, and then the user calls `mint()` function to mint 10 `clevcvx token` for him. After a period of time, the user calls the `repay()` function to repay the debt in the form of WETH (underlying token). According to the code logic, the user needs to use 10 WETH to repay the 10 clevcvx tokens he has borrowed. Obviously, this code logic does not consider the actual value of the WEH (underlying token) with the clevcvx token (debt token) and instead defaults to an equivalent exchange. Such an approach to debt repayment may result in a mismatch between the actual value of the underlying token being repaid and the theoretical value to be repaid.

```
function repay(
    address _underlyingToken,
    address _recipient,
    uint256 _amount
  ) external override nonReentrant {
```

```
    ......
    // 2. check debt and update debt
    {
      int256 _debt = _userInfo.totalDebt;
      require(_debt > 0, "CLever: no debt to repay");
      uint256 _scale = 10**(18 -
IERC20Metadata(_underlyingToken).decimals());
      uint256 _maximumAmount = uint256(_debt) / _scale;
      if (_amount > _maximumAmount) _amount = _maximumAmount;
      uint256 _debtPaid = _amount * _scale;
      _userInfo.totalDebt = int128(_debt - int256(_debtPaid));
// safe to do cast
    }

    // 3. take fee and transfer token to Furnace
    FeeInfo memory _feeInfo = feeInfo;
    if (_feeInfo.repayPercentage > 0) {
      uint256 _fee = (_amount * _feeInfo.repayPercentage) /
FEE_PRECISION;

 IERC20Upgradeable(_underlyingToken).safeTransferFrom(msg.send
er, _feeInfo.platform, _fee);
    }
    address _furnace = furnace;

 IERC20Upgradeable(_underlyingToken).safeTransferFrom(msg.send
er, _furnace, _amount);
    IMetaFurnace(_furnace).distribute(address(this),
_underlyingToken, _amount);

    emit Repay(_recipient, _underlyingToken, _amount);
  }
```

Status

This issue has been discussed, and the team has confirmed that it will not happen in practice.

4.3.2 Discussion of the parameter `expectedUnderlyingTokenAmount`.

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

Description

The user calls the `deposit()` function to deposit funds into the specified `strategyIndex` strategy, which can be either `underlyingToken` or `yieldToken`. Specifically, when `underlyingToken` is CRV token, this function will call the `deposit()` function under the `AladdinCRVStrategy` contract to deposit the user's CRV token into the Convex protocol. At the same time, the `AladdinCRVStrategy` contract will receive its share of aCRV tokens. Next, the internal function `_updateActiveBalance()` updates the parameter `expectedUnderlyingTokenAmount`. We find that the amount of cvxcrv token is recorded under the parameter `expectedUnderlyingTokenAmount`, whereas the underlying token is CRV token, so we need to confirm what the parameter `expectedUnderlyingTokenAmount` is the actual meaning of the parameter `expectedUnderlyingTokenAmount`.

```
function deposit(
    uint256 _strategyIndex,
    address _recipient,
    uint256 _amount,
    uint256 _minShareOut,
    bool _isUnderlying
  ) external override nonReentrant
onlyActiveStrategy(_strategyIndex) returns (uint256 _shares) {
```

```
    require(_amount > 0, "CLever: deposit zero amount");

    YieldStrategyInfo storage _yieldStrategy =
yieldStrategies[_strategyIndex];
    UserInfo storage _userInfo = userInfo[_recipient];

    // 1. transfer token to yield strategy
    address _strategy = _yieldStrategy.strategy;

    //@audit assume that underlyingToken is CRV token
    address _token = _isUnderlying ?
_yieldStrategy.underlyingToken : _yieldStrategy.yieldToken;
    {
      //@audit transfer funds to the corresponding strategy
contract
      uint256 _beforeBalance =
IERC20Upgradeable(_token).balanceOf(_strategy);
      IERC20Upgradeable(_token).safeTransferFrom(msg.sender,
_strategy, _amount);
      _amount =
IERC20Upgradeable(_token).balanceOf(_strategy).sub(_beforeBala
nce);
    }
    // @note reuse `_amount` to store the actual yield token
deposited.
    _amount = IYieldStrategy(_strategy).deposit(_recipient,
_amount, _isUnderlying);

    ......

    // 5. update yield strategy info
    _yieldStrategy.totalShare = _totalShare.add(_shares);
    _updateActiveBalance(_strategyIndex, int256(_amount));

    ......
  }
```

```
function _updateActiveBalance(uint256 _strategyIndex, int256
_delta) internal {
    uint256 _activeYieldTokenAmount =
yieldStrategies[_strategyIndex].activeYieldTokenAmount;
    uint256 _expectedUnderlyingTokenAmount =
yieldStrategies[_strategyIndex].expectedUnderlyingTokenAmount;

    //@audit the _rate indicates the amount of cvxcrv tokens
corresponding to the unit acrv token
    uint256 _rate =
IYieldStrategy(yieldStrategies[_strategyIndex].strategy).under
lyingPrice();

    if (_delta > 0) {
      _activeYieldTokenAmount =
_activeYieldTokenAmount.add(uint256(_delta));
      _expectedUnderlyingTokenAmount =
_expectedUnderlyingTokenAmount.add(uint256(_delta).mul(_rate)
/ PRECISION);
    } else {
     ......
    }

    yieldStrategies[_strategyIndex].activeYieldTokenAmount =
_activeYieldTokenAmount;

 yieldStrategies[_strategyIndex].expectedUnderlyingTokenAmount
= _expectedUnderlyingTokenAmount;
  }


//@audit located in AladdinCRVStrategy.sol
function deposit(
    address,
    uint256 _amount,
    bool _isUnderlying
  ) external override onlyOperator returns (uint256
_yieldAmount) {
    if (_isUnderlying) {
```

```
        _yieldAmount =
IAladdinCRV(aCRV).depositWithCRV(address(this), _amount);
    } else {
      _yieldAmount = _amount;
    }
  }

//@audit located in AladdinCRVStrategy.sol
function underlyingPrice() external view override returns
(uint256) {
    uint256 _totalUnderlying =
IAladdinCRV(aCRV).totalUnderlying();
    //@audit total supply of acrv token
    uint256 _totalSupply = IERC20(aCRV).totalSupply();
    return (_totalUnderlying * 1e18) / _totalSupply;
  }
```

Status

This issue has been discussed. The team confirmed that one cvxCRV token is treated as one CRV token.

4.3.3 Add judgement on the parameter `_share` to optimize the code structure.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	More gas consumption	Discussed

Description

Add judgment on the parameter `_share` value to skip unnecessary calculations when `_share == 0`, which could optimize the code structure and reduce gas consumption.

```
function _updateReward(uint256 _strategyIndex, address
_account) internal {
```

```
    UserInfo storage _userInfo = userInfo[_account];
    YieldStrategyInfo storage _yieldStrategyInfo =
yieldStrategies[_strategyIndex];

    uint256 _share = _userInfo.share[_strategyIndex];

    ......
    if (_accRewardPerSharePaid < _accRewardPerShare) {
      uint256 _scale = 10**(18 -
IERC20Metadata(_token).decimals());
      uint256 _rewards = (_share.mul(_accRewardPerShare -
_accRewardPerSharePaid) / PRECISION).mul(_scale);
      _userInfo.totalDebt -=
SafeCastUpgradeable.toInt128(SafeCastUpgradeable.toInt256(_rew
ards));
      _userInfo.accRewardPerSharePaid[_strategyIndex][_token]
= _accRewardPerShare;
    }

    // 2. update extra rewards
    uint256 _length =
_yieldStrategyInfo.extraRewardTokens.length;
    for (uint256 i = 0; i < _length; i++) {
      ......
      if (_accRewardPerSharePaid < _accRewardPerShare) {
        uint256 _rewards = _share.mul(_accRewardPerShare -
_accRewardPerSharePaid) / PRECISION;
        _userInfo.pendingRewards[_strategyIndex][_token] +=
_rewards;
        _userInfo.accRewardPerSharePaid[_strategyIndex]
[_token] = _accRewardPerShare;
      }
    }
  }
```

Suggestion

The corresponding modifications are as follows.

```
function _updateReward(uint256 _strategyIndex, address
_account) internal {
    UserInfo storage _userInfo = userInfo[_account];
    YieldStrategyInfo storage _yieldStrategyInfo =
yieldStrategies[_strategyIndex];

    uint256 _share = _userInfo.share[_strategyIndex];

    ......
    if (_accRewardPerSharePaid < _accRewardPerShare) {
      //@audit add judgment about _share
      if(_share > 0){
      uint256 _scale = 10**(18 -
IERC20Metadata(_token).decimals());
      uint256 _rewards = (_share.mul(_accRewardPerShare -
_accRewardPerSharePaid) / PRECISION).mul(_scale);
      _userInfo.totalDebt -=
SafeCastUpgradeable.toInt128(SafeCastUpgradeable.toInt256(_rew
ards));
      }
      _userInfo.accRewardPerSharePaid[_strategyIndex][_token]
= _accRewardPerShare;
    }

    // 2. update extra rewards
    uint256 _length =
_yieldStrategyInfo.extraRewardTokens.length;
    for (uint256 i = 0; i < _length; i++) {
      ......
      if (_accRewardPerSharePaid < _accRewardPerShare) {
        //@audit add judgment about _share
        if(_share > 0){
        uint256 _rewards = _share.mul(_accRewardPerShare -
_accRewardPerSharePaid) / PRECISION;
```

```
        _userInfo.pendingRewards[_strategyIndex][_token] +=
_rewards;
        }
        _userInfo.accRewardPerSharePaid[_strategyIndex]
[_token] = _accRewardPerShare;
      }
    }
  }
```

Status

The team has confirmed and plans to improve it in the next release.

5. Conclusion

After auditing and analyzing the CLever extension 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