Security Audit Report

ClevUSD Strategy by AladdinDAO

SECBIT

September 28, 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 development team has added a new cleverUSD revenue strategy to the existing clever protocol. In addition, the development team has also updated the token vesting contract. SECBIT Labs conducted an audit from September 14 to September 28, 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 on the logic of collecting released tokens.	Info	Discussed
Design & Implementation	4.3.2 Potential risk of DOS attacks.	Low	Fixed
Design & Implementation	4.3.3 Discussion on the logic of preventing price manipulation.	Medium	Fixed
Design & Implementation	4.3.4 Discussion of potential price-manipulation attacks.	Medium	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the ClevUSD strategy contract is shown below:

- Smart contract code
  - initial review commit *dcef2fe*
  - final review commit *8761cb3*

2.2 Contract List

The following content shows the contracts included in the Aladdin ClevUSD strategy, which the SECBIT team audits:

Name	Lines	Description
AllInOneGateWay.sol	185	A generic interface contract for the clever protocol that integrates the asset vaulting functionality.
ConcentratorStrategy.sol	175	Concentrator strategy for CLever protocol.
CurveBasePoolChecker.sol	59	This contract checks whether the specified basepool under Curve protocol has been manipulated in price.
CurveMetaPoolChecker.sol	46	This contract checks whether the specified metapool under Curve protocol has been manipulated in price.
CLeverCVXLocker.sol	628	This contract helps users automatically lock CVX tokens and provide clevCVX token lending services.
Vesting.sol	94	A token vesting contract.

Notice: `ConcentratorStrategy.sol` *and* `CLeverCVXLocker.sol` *have been modified from an old audited version. We mainly focus on the modified parts.*

3. Contract Analysis

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

3.1 Role Classification

Two key roles in the ClevUSD strategy protocol are 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 transferring the governance account.
- Common Account
    - Description

      Users participate in the Aladdin ClevUSD protocol.
    - Authority
        - Deposit underlying tokens and receive rewards
    - Method of Authorization

      No authorization required

3.2 Functional Analysis

The ClevUSD method expands the revenue strategy for the Aladdin protocol. 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:

ConcentratorStrategy

This contract allows the user to deposit funds into a specified gauge and receive a corresponding share. With this share, the user will obtain profits.

The main functions in `ConcentratorStrategy` are as below:

- `deposit()`

This function allows the user to deposit yield tokens to the vault. At the same time, he will get share tokens.

- `withdraw()`

  The user withdraws a specified number of token assets and burns the corresponding share of share tokens.

- `harvest()`

  Users can call this function to harvest rewards.

CurveBasePoolChecker & CurveMetaPoolChecker

The function of these two contracts is to check that the liquidity pool of the Curve protocol used by the user is not maliciously manipulated in price to prevent a potential sandwich attack.

The main functions in `CurveBasePoolChecker` and `CurveMetaPoolChecker` is as below:

- `check()`

  This function first revises the balance of each asset in the pool. It then calculates the ratio of the maximum to the minimum number of assets in the pool to predict whether the price of the assets has been manipulated.

Vesting

Those tokens purchased by a user will be transferred to this contract and then released linearly at a scheduled time.

The main functions of `Vesting` are as below:

- `claim()`

  This function allows users to claim tokens that have been released.

- `newVesting()`

  This function records the number of tokens purchased by the user and key information such as the start time and end time of release.

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 on the logic of collecting released tokens.

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

Description

The user calls the `claim()` function to retrieve the released token. According to the logic of the current contract, tokens purchased by the user through the `Tokensale` contract will be released to the user linearly on a deferred basis. Each token purchased by the user will be recorded in a different structure. When the user calls the `claim()` function to claim the currently released tokens, the function will iterate through the array from the beginning and calculate the total number of tokens the user can claim. As time grows, some of the funds in the array may have been claimed. Assuming that the first tokens purchased by the user have already been claimed, but the code still counts from the first array of structures, this will waste unnecessary gas and increase the cost. It is essential to consider whether the code structure needs to be optimized, considering the frequency of use of this contract and the cost of implementation for the user.

```
function claim() external returns (uint256 _claimable) {
    uint256 _length = vesting[msg.sender].length;
    //@audit the reward will be calculated each time
    //       starting from the first array of structures
    for (uint256 i = 0; i < _length; i++) {
      VestState memory _state = vesting[msg.sender][i];

      uint256 _vested = _getVested(_state, block.timestamp);
      vesting[msg.sender][i].claimedAmount = uint128(_vested);

      _claimable += _vested - _state.claimedAmount;
    }

    IERC20(token).safeTransfer(msg.sender, _claimable);

    emit Claim(msg.sender, _claimable);
}
```

Status

This issue has been discussed. In conjunction with the actual usage scenario, the development team decided not to adjust the structure of the code.

4.3.2 Potential risk of DOS attacks.

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

Description

The `newVesting()` function generates `vesting` information for the user, which records the number of tokens purchased by the user and the linear release period. Note that the `newVesting()` function is an `external` type function with no privileges check, so anyone can call it directly, which could raise a potential DOS risk. Suppose malicious userA calls the `newVesting()` function multiple times to generate additional arrays of `VestState` structures for normal userB, significantly increasing the length of the `vesting[B.address]` array. When userB calls the`claim()` function to collect the released token, the gas consumption increases due to the malicious array generated by userA. In an extreme case, when the `vesting` array generated by the malicious user is too long, it will fail for userB to call the `claim()` function because it runs out of gas.

```
function claim() external returns (uint256 _claimable) {
    uint256 _length = vesting[msg.sender].length;
    for (uint256 i = 0; i < _length; i++) {
     ......
    }

    IERC20(token).safeTransfer(msg.sender, _claimable);

    emit Claim(msg.sender, _claimable);
  }

function newVesting(
    address _recipient,
    uint128 _amount,
    uint64 _startTime,
    uint64 _endTime
  ) external {
    require(_startTime < _endTime, "Vesting: invalid timestamp");

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

    uint256 _index = vesting[_recipient].length;
    vesting[_recipient].push(
      VestState({
        vestingAmount: _amount,
        claimedAmount: 0,
        startTime: _startTime,
        endTime: _endTime,
        cancleTime: uint64(block.timestamp)
      })
    );

    emit Vest(_recipient, _index, _amount, _startTime, _endTime);
  }
```

Suggestion

Add permission check for the `newVesting()` function to prevent malicious users from calling it.

Status

The development team adopted our advice and fixed this issue in commit 8761cb3.

4.3.3 Discussion on the logic of preventing price manipulation.

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

Description

The following example uses a basepool consisting of USDT, WBTC, and WETH. In practice, the basepool/metapool is anchored to the same token in most cases, but there still exists a risk of bypassing the `check()` function. An attacker can manipulate the price while the check formula always holds by trading the minimum decimal token for the maximum token.

The `check()` function is designed to check for price manipulation in the corresponding basepool in Curve protocol to prevent the `Operator` from losing money when exchanging tokens. It queries the balance of the assets in the basepool and calculates the ratio of the maximum number of assets to the minimum number of assets in the pool. When the ratio is less than the value preset by the administrator, no price manipulation is considered to have occurred. Such a design does not seem to serve the purpose of detecting whether prices are being manipulated. The above judgment is based on the following inequality.

$$\frac{\_maxBalance}{\_minBalance} \leq \frac{maxMultiple}{PRECISION} \tag{1}$$

The `maxMultiple` parameter in the inequality is set by the administrator, and the `PRECISION` parameter is a constant, so the right-hand side of the inequality can be considered constant in a specific scenario. In this case, the detection can be bypassed by controlling the ratio of the parameters on the left side of the inequality.

Specifically, take the USDT+wBTC+WETH as an example. The three assets in the pool are currently almost identical in value, each accounting for one-third of the pool's assets. However, the number of assets varies significantly due to the difference in decimals and the value of each asset. The current pool has about 5.69×10^{13} of USDT, 3×10^{11} of wBTC, and 4.2×10^{22} of wETH, with wETH far outnumbering the other two assets. The attacker can deposit a huge value of wBTC through flash loan and exchange it for a large amount of wETH, which in this case has seriously affected the normal asset exchange price. Based on the actual situation, the following inequalities are available.

$$\left(\frac{\_maxBalance'}{\_minBalance'} = \frac{3.13 \times 10^{22}}{4 \times 10^{11}}\right) < \left(\frac{\_maxBalance}{\_minBalance} = \frac{4.2 \times 10^{22}}{3 \times 10^{11}}\right) \leq \frac{maxMultiple}{PRECISION} \quad (2)$$

At this point, the price manipulation still successfully passes the `check()` function. Assuming that the `UnderlyingToken` is wBTC and the `yieldToken` is wETH, there is still the risk of a sandwich attack manipulating the price.

```
function check(address _token) external view override returns (bool) {
    // find pool address
    address _pool = ICurvePoolRegistry(REGISTRY).get_pool_from_lp_token(_token);
    if (_pool == address(0)) {
      _pool = _token;
    }

    // find min/max balance
    uint256 _maxBalance = 0;
    uint256 _minBalance = uint256(-1);
    for (uint256 i = 0; ; i++) {
      // vyper is weird, some use `int128`
      //@audit Read the amount of funds in the pool
      try ICurveBasePool(_pool).balances(i) returns (uint256 _balance) {
        if (_balance > _maxBalance) _maxBalance = _balance;
        if (_balance < _minBalance) _minBalance = _balance;
      } catch {
        try ICurveBasePool(_pool).balances(int128(i)) returns (uint256 _balance)
{
          if (_balance > _maxBalance) _maxBalance = _balance;
          if (_balance < _minBalance) _minBalance = _balance;
        } catch {
          break;
        }
      }
    }

    // _maxBalance / _minBalance <= maxMultiple / PRECISION
    return _maxBalance.mul(PRECISION) <= maxMultiple.mul(_minBalance);
  }
```

Status

After discussion, the development team revised the number of tokens according to the decimals of the different tokens so that they were on the same standard scale. This solution is effective in preventing potential sandwich attacks in commit 8761cb3.

4.3.4 Discussion of potential price-manipulation attacks.

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

Description

Similar to issue 4.3.3, the curve's metapool pool involves a wider variety of funds, and here it is necessary to verify the specific `underlyingToken`, `yieldToken` parameter. Also, it should be debated whether the price check logic is reliable.

```
function check(address _token) external view override returns (bool) {
    // find pool address
    address _pool = ICurvePoolRegistry(REGISTRY).get_pool_from_lp_token(_token);
    if (_pool == address(0)) {
      _pool = _token;
    }

    // check base pool
    //@audit check for price manipulation in the basepool
    if
(!IPriceChecker(baseChecker).check(ICurveBasePool(_pool).coins(uint256(1)))) {
      return false;
    }

    // find min/max balance
    uint256 _maxBalance = 0;
    uint256 _minBalance = uint256(-1);
    for (uint256 i = 0; i < 2; i++) {
      uint256 _balance = ICurveBasePool(_pool).balances(i);
      if (_balance > _maxBalance) _maxBalance = _balance;
      if (_balance < _minBalance) _minBalance = _balance;
    }

    // _maxBalance / _minBalance <= maxMultiple / PRECISION
    // @audit check for price manipulation in the metapool
    return _maxBalance.mul(PRECISION) <= maxMultiple.mul(_minBalance);
  }
```

Status

The same solution as for issue 4.3.3, the development team fixed this issue in commit 8761cb3.

5. Conclusion

After auditing and analyzing the Aladdin ClevUSD strategy 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.