Security Audit Report

TokenSale Contract 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 CLever protocol helps users automatically lock CVX tokens into the Convex protocol. As an ancillary part, the TokenSale contract provides token purchase service. At the end of the sale phase, part of the quota tokens purchased by the user will be sent directly to the user, and the rest will be released linearly over time. SECBIT Labs conducted an audit from April 15 to April 27, 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 Mapping `isSupported` function discussion.	Info	Fixed
Design & Implementation	4.3.2 The amount of quota tokens purchased by whitelisted users is not set.	Info	Fixed
Design & Implementation	4.3.3 Adjust the sequence of operations to improve the accuracy of the calculation.	Info	Fixed
Design & Implementation	4.3.4 Discussion of function `updateVesting()`.	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 contract is shown below:

- Smart contract code
  - initial review commit *b0dd8fc*
  - final review commit *9018118*

2.2 Contract List

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

Name	Lines	Description
TokenSale.sol	229	Offers quota token sale services.

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 TokenSale: Governance Account and Common Account.

- Governance Account
    - Description

      Contract administrator
    - Authority
        - Update sale start time and duration
        - Update token sale price
        - Update vesting information
        - Update supported tokens
        - 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 TokenSale.
    - Authority
        - Purchase quota token
        - Claim purchased quota token
    - Method of Authorization

      No authorization required

3.2 Functional Analysis

The TokenSale contract provides a quota token selling service. The SECBIT team conducted a detailed audit of this contract, and the essential functions in the contract are as follows.

TokenSale

This contract provides quota tokens selling service. The selling phase is divided into the whitelist pre-sale and the public sale. In the whitelist pre-sale phase, only whitelisted users can purchase quota tokens, and each whitelisted user is limited to the number of tokens they can purchase during the pre-sale phase. At the end of the pre-sale phase, the public sale of quota tokens begins, and any user is allowed to purchase them at this time. At the end of the pre-sale phase, users will be able to retrieve some of their purchased quota tokens, and the rest will be sent to them in a linear release.

The main functions in `TokenSale` are as below:

- `buy()`

  This function allows the user to purchase some quota tokens in the contract using a supported base token.

- `claim()`

  This function allows the user to claim the purchased quota 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 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 Mapping `isSupported` function discussion.

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

Description

The mapping `isSupported` is used to determine whether the `token` paid by the user is allowed to purchase the `quota token`. By default, the result of `isSupported[A]` is `false` for any address. There is no function in the current token sale contract that handles the setting of the mapping `isSupported`, so the relevant require condition holds in any case. The mapping `isSupported` has no practical meaning.

```
mapping(address => bool) public isSupported;

function buy(
    address _token,
    uint256 _amountIn,
    uint256 _minOut
  ) external payable nonReentrant returns (uint256) {
    require(_amountIn > 0, "TokenSale: zero input amount");

    // 1. check supported token
    // @audit hold constantly
    require(!isSupported[_token], "TokenSale: token not
support");

    // 2. check sale time
    SaleTimeData memory _saleTime = saleTimeData;
```

```
    ......
}
```

Status

The team fixed the issue themselves before our feedback in commit 58135da.

4.3.2 The amount of quota tokens purchased by whitelisted users is not set.

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

Description

The quota tokens are sold with a pre-purchase limit for each whitelisted user. No function in the current code sets the number of quota tokens that a whitelisted user can purchase. In this case, no user can buy quota tokens during `whitelistSaleTime`, and the whitelist setting is meaningless. Therefore, it is necessary to add functions to handle the number of quota tokens purchased by whitelisted users.

```
mapping(address => uint256) public whitelistCap;

function buy(
    address _token,
    uint256 _amountIn,
    uint256 _minOut
  ) external payable nonReentrant returns (uint256) {
    require(_amountIn > 0, "TokenSale: zero input amount");

    // 1. check supported token
    require(!isSupported[_token], "TokenSale: token not
support");

    // 2. check sale time
```

```
    SaleTimeData memory _saleTime = saleTimeData;
    require(block.timestamp >= _saleTime.whitelistSaleTime,
"TokenSale: sale not start");
    require(block.timestamp <= _saleTime.publicSaleTime +
_saleTime.saleDuration, "TokenSale: sale ended");

    // 3. determine account sale cap
    uint256 _cap = cap;
    uint256 _totalSold = totalSold;
    uint256 _saleCap = _cap.sub(_totalSold);
    require(_saleCap > 0, "TokenSale: sold out");

    uint256 _userCap;
    if (block.timestamp < _saleTime.publicSaleTime) {
      //@audit no purchase limit
      _userCap =
whitelistCap[msg.sender].sub(shares[msg.sender]);
      if (_userCap > _saleCap) {
        _userCap = _saleCap;
      }
    } else {
      _userCap = _saleCap;
    }
    ......
  }
```

Status

The development team fixed the issue themselves before our feedback in commit 8006992.

4.3.3 Adjust the sequence of operations to improve the accuracy of the calculation.

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

Description

We can improve the accuracy of the calculation by performing the multiplication operation first and then the division operation, provided that the result of the calculation does not overflow.

```
function getPrice() public view returns (uint256) {
    PriceData memory _data = priceData;
    uint256 _totalSold = totalSold;

    return
      RATIO_PRECISION
        .mul(_data.variation)
        .add(_totalSold.mul(_data.upRatio))
        .div(_data.variation)
        .mul(_data.initialPrice)
        .div(RATIO_PRECISION);
  }
```

Suggestion

If the calculation does not overflow, the order of calculation can be adjusted as follows.

```
function getPrice() public view returns (uint256) {
    PriceData memory _data = priceData;
    uint256 _totalSold = totalSold;

    return
      RATIO_PRECISION
        .mul(_data.variation)
        .add(_totalSold.mul(_data.upRatio))
        .mul(_data.initialPrice)
        .div(_data.variation)
        .div(RATIO_PRECISION);
  }
```

Status

The team fixed this issue in commit f00f063.

4.3.4 Discussion of function `updateVesting()`.

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

Description

When a purchaser retrieves their purchased `quota token` by executing the `claim()` function, a certain percentage of the `quota token` is locked into the vesting contract and released linearly. The function `updateVesting()` is used to set the parameters related to the linear release of `quota token` from the vesting contract and can be called at any time. Given that calling the `updateVesting()` function after the start of a `claim` may cause the number and rate of linear release of quota tokens to vary between users, it is important to clarify whether the `updateVesting()` function can be called after the start of a `claim`.

```
function updateVesting(
    address _vesting,
    uint32 _vestRatio,
    uint64 _duration
  ) external onlyOwner {
    require(_vestRatio <= RATIO_PRECISION, "TokenSale: ratio
too large");
    require(_duration > 0, "TokenSale: zero duration");

    //@audit does it need to be restricted ?
    vestingData = VestingData(_vesting, _vestRatio,
_duration);

    emit UpdateVesting(_vesting, _vestRatio, _duration);
  }
```

Status

This issue has been discussed.

5. Conclusion

After auditing and analyzing the TokenSale 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