Security Audit Report

AladdinDAO V2

SECBIT

December 23, 2021

1. Introduction

The AladdinDAO is a decentralized network to shift crypto investments from venture capitalists to the wisdom of crowds through collective value discovery. AladdinDAO (V2) adopts a new model for boule members to manage protocol assets and mining token incentives under PCV. SECBIT Labs conducted an audit from December 1st to December 23nd, 2021, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the AladdinDAO (V2) 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 Traversal unbounded array in a function may cause failure.	Low	Fixed
Design & Implementation	4.3.2 Discussion on the speed of ALD token issuance.	Discussion	Discussed
Design & Implementation	4.3.3 Incorrect use of the `!=` affects the normal logic of the `distribute()` function.	Medium	Fixed
Design & Implementation	4.3.4 In the `ALDDaoV2` contract, there is unclear logic for the deposit of the ALD token into the `Staking` contract.	Discussion	Discussed

Design & Implementation	4.3.5 Wrong initial condition setting can lead to incorrect calculation results.	Medium	Fixed
Design & Implementation	4.3.6 Incorrect use of the `blockNumber` parameter as `epochNumber` parameter will result in incorrect calculation results.	Medium	Fixed
Design & Implementation	4.3.7 There is a risk of price manipulation when calling Uniswap V2 to calculate the value of the LP token.	Medium	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about AladdinDAO (V2) contract is shown below:

- Project website
  - https://aladdin.club/
- Smart contract code
  - https://github.com/AladdinDAO/aladdin-contracts/tree/contracts-v2
  - initial review commit *605abea*
  - final review commit *f446af3*

2.2 Contract List

The following content shows the contracts included in the AladdinDAO (V2), which the SECBIT team audits:

Name	Lines	Description
ALDDaoV2.sol	87	A specific contract for processing DAO members' rewards.
Airdrop.sol	69	An airdrop award contract for project participants.
DAODistributor.sol	49	A specific contract for distributing ALD reward for DAO to different recipients

Keeper.sol	70	A contract to rebase ALD tokens and harvest vault rewards.
Treasury.sol	228	A contract that holds user's bond assets, such as ether, LP token, etc.
DirectBondDepositor.sol	81	A contract to bond ALD token by depositing the specified token.
RewardBondDepositor.sol	335	A contract to handle vault rewards.
ChainlinkPriceOracle.sol	27	Read the specified token price from Chainlink oracle.
UniswapTWAPOracle.sol	178	Collect the moving price average in UniswapV2.
UniswapV2PairPriceOracle.sol	75	Get the price of the LP token from UniswapV2.
UniswapV2PriceOracle.sol	53	As a complement, it will get a price from UniswapV2 if Chainlink oracle does not provide the token price.
Distributor.sol	42	Distribute ALD token reward to `Staking` contract.
Staking.sol	534	The core contract handles the stake of ALD token and the redemption of unlocked xALD token by users.
WrappedXALD.sol	36	This contract implements the swap between the xALD token and wxALD token.
XALD.sol	156	The xALD token acts as a bridge between the ALD token and the wxALD token.

MultipleRewardsVaultBase.sol	140	A base contract that handles multiple reward tokens.
SingleRewardVaultBase.sol	123	A base contract that handles single reward token.
VaultBase.sol	33	A contract is used to modify the `bondPercentage` parameter.
BaseConvexVault.sol	66	This abstract contract implements functions such as `deposit()`, `withdraw()` and `harvest()`.

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 AladdinDAO (V2): Governance Account and Common Account.

- Governance Account
  - Description

    Contract administrator
  - Authority
    - Update basic parameters
    - Update moving price averages in the past
    - Transfer ownership
  - Method of Authorization

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

- Common Account
  - Description

    Users participate in AladdinDAO (V2)
  - Authority
    - Stake ALD token by Staking contract
    - Deposit bond token by DirectBondDepositor contract
    - Deposit vault tokens by vault related contracts
  - Method of Authorization

    No authorization required

3.2 Functional Analysis

The AladdinDAO (V2) establishes a new paradigm for token allocation for liquidity mining. 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 several parts:

Treasury

The `Treasury` contract is used to hold bond tokens. It implements the core logic of the bond token deposit to the AladdinDAO (V2).

The main functions in `Treasury` are as below:

- `valueOf()`

  Get the USD value of the specified amount of tokens.
- `deposit()`

  This function is the core implementation for depositing bond tokens to mint ALD tokens. It can only be called by specific contracts.

DirectBondDepositor

The `DirectBondDepositor` contract provides a user call interface to handle user requests directly.

The main functions in `DirectBondDepositor` are as below:

- `deposit()`

  Users can call this function directly to bond ALD tokens.

RewardBondDepositor

This contract processes the rewards earned by users from Convex finance and converts these rewards into ALD tokens for locking. In addition, this contract also implements the ability to rebase ALD tokens.

The main functions in `RewardBondDepositor` are as below:

- `notifyRewards()`

  This function records the rewards that the user has received from Convex finance.

- `rebase()`

  This function converts the user's reward token into an ALD token and locks it. At the same time, this function mints additional ALD tokens as a fixed percentage of the total current ALD token supply as a reward for the user.

Staking

This contract focuses on the deposit of ALD tokens and the redeem of unlocked tokens.

The main functions in `Staking` are as below:

- `stake()`

The user calls this function to deposit a specified number of ALD tokens. Then, wxALD tokens are calculated and released to the user linearly based on time.

- `unstake()`

  The user swaps xALD tokens for ALD tokens and transfers them to the recipient address.

- `bondFor()`

  This function handles tokens deposited by the user using the `DirectBondDepositor` contract. These specific tokens are first converted into ALD tokens, then transferred to the `Staking` contract. The corresponding information is recorded via the `bondFor()` function.

- `rewardBond()`

  This function is used to process the rewards received by the project from Convex finance, which will be recorded in the `rewardBondLocks` array.

- `rebase()`

  This function implements the logic for incrementing ALD tokens amount. The new minted ALD tokens are locked under the `Staking` contract, and these ALD tokens will be used as a reward for the user who holds xALD tokens.

- `redeem()`

  The user redeems the unlocked wxALD tokens via this function. These wxALD tokens will be transferred to the user as xALD tokens or ALD tokens, depending on the user's preferences.

WrappedXALD

The wxALD token is a standard ERC20 token that users can swap for the wxALD token using the xALD token.

The main functions in `WrappedXALD` are as below:

- `wrap()`

Allows users to exchange their xALD tokens for wxALD tokens.

- `unwrap()`

  Allows users to exchange their wxALD tokens for xALD tokens.

XALD

The xALD token is a token that changes dynamically with time. The longer a user holds xALD tokens, the more ALD tokens they will receive. As a bridge, xALD tokens can be exchanged for wxALD tokens or ALD tokens.

The main functions in `XALD` are as below:

- `stake()`

  This function mints the corresponding xALD token based on the amount of ALD tokens provided by the staking contract.

- `unstake()`

  Burn the corresponding amount of xALD tokens to redeem the ALD tokens.

- `rebase()`

  Adjust the value of the `totalSupply` parameter, which will increase the amount of ALD tokens per xALD token.

MultipleRewardsVaultBase

This contract focuses on vault rewards. The main functions in `MultipleRewardsVaultBase` are as below:

- `deposit()`

  This function allows the user to deposit the base token into the vault, eventually being transferred to Convex finance to receive the earnings.

- `withdraw()`

  This function allows the user to withdraw the principal that the user has deposited.

- `claim()`

  Users can claim pending rewards from the vault by this function.

- `harvest()`

  Anyone can call this function to retrieve the earnings in Convex finance. The earnings will be divided into two parts. One will remain in the contract, and the other will be converted into ALD tokens so that that project participants will receive both rewards.

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 bug, 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 Traversal unbounded array in a function may cause failure.

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

Description

The `accountEpochShares` parameter is used to record the user's share in a specific `epoch` interval under the `vault`. This share information is recorded in the `AccountEpochShare` array, and the length of this array grows as the number of user operations increases. The `getAccountRewardShareSince()` function is used to read the user's share in each epoch from the start of the specified `_epoch` index to the latest epoch, which is recorded in the `_shares` array. The code exploits the user's share array when performing the share calculation. As the epoch and user operations increase, the length of this array will gradually increase. It may exceed the gas limit, eventually causing the function to fail.

```
// located in RewardBondDepositor.sol
struct AccountEpochShare {
    uint32 startEpoch; // include
    uint32 endEpoch; // not inclued
    uint192 totalShare;
  }

mapping(address => mapping(address => AccountEpochShare[]))
private accountEpochShares;

function getAccountRewardShareSince(
```

```
    uint256 _epoch,
    address _user,
    address _vault
  ) external view override returns (uint256[] memory) {
    ......

    _getRecordedAccountRewardShareSince(_epoch, _user, _vault,
_shares);
   _getPendingAccountRewardShareSince(_epoch, _user, _vault,
_shares);

    return _shares;
  }

function _getRecordedAccountRewardShareSince(
    uint256 _epoch,
    address _user,
    address _vault,
    uint256[] memory _shares
  ) internal view {
    AccountEpochShare[] storage _accountEpochShares =
accountEpochShares[_user][_vault];
    uint256 length = _accountEpochShares.length;

    Epoch memory _cur = currentEpoch;
    Epoch memory _now = epoches[0];
    Epoch memory _next;
    for (uint256 i = 0; i < length; i++) {

      ......

      for (uint256 j = _start; j < _epochShare.endEpoch; j++)
{
        if (_epochShare.endEpoch == _epochShare.startEpoch +
1) {
          _shares[j - _epoch] = _epochShare.totalShare;
        } else {
          if (_epochShare.endEpoch == _cur.epochNumber) {
            _next = _cur;
```

```
            } else {
              _next = epoches[j];
            }
            _shares[j - _epoch] =
uint256(_epochShare.totalShare).mul(_next.epochLength).div(blo
cks);
          }
        }
      }
    }
```

Status

The developer team avoided these problems by slicing the information in the array to read the specified fragment in commit a2490d8.

4.3.2 Discussion on the speed of ALD token issuance.

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

Description

The current code allows the keeper to call the `rebase()` function per epoch (about 6400 blocks) to increase total supply by a fixed percentage of the current ALD token issuance. The default increment rate in the contract is `rewardRate = 3e15`, at which the total ALD token issuance after 100 days will be 1.35 times relative to the current total issuance. Considering that ALD tokens are also minted when users deposit the underlying assets (ETH, WBTC, etc.), this will further increase the total ALD token issuance. Therefore the ALD token may increase even faster. The issue of ALD token issuance needs to be considered to maintain the project's stability.

```
// located in staking.sol
function rebase() external override notPaused {
```

```
    require(rewardBondDepositor == msg.sender, "Staking: not
approved");

    if (distributor != address(0)) {
      uint256 _pool = IERC20(ALD).balanceOf(address(this));
      IDistributor(distributor).distribute();
      uint256 _distributed =
IERC20(ALD).balanceOf(address(this)).sub(_pool);

      (uint256 epochNumber, , , ) =
IRewardBondDepositor(rewardBondDepositor).currentEpoch();
      IXALD(xALD).rebase(epochNumber, _distributed);
    }
  }

// located in Distributor.sol
function distribute() external override {
    require(msg.sender == staking, "Distributor: not
approved");

    uint256 _reward = nextRewardAt(rewardRate);
    ITreasury(treasury).mintRewards(staking, _reward);
  }

function nextRewardAt(uint256 _rate) public view returns
(uint256) {
    return
IERC20(ald).totalSupply().mul(_rate).div(PRECISION);
  }
```

Status

This issue has already been discussed. The development team deals with this issue by adjusting the reward rate dynamically.

4.3.3 Incorrect use of the `!=` affects the normal logic of the `distribute()` function.

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

Description

The `distribute()` function distributes rewards for given users. It should require the user address array to be equal to the length of the corresponding money array. The `require` condition in the code incorrectly uses the not equal sign, which interferes with the logic of the normal code.

```
// located in Airdrop .sol
function distribute(address[] memory _users, uint256[] memory
_amounts) external {
    require(_users.length != _amounts.length, "Airdrop: length
mismatch");

    uint256 _totalAmount;
    for (uint256 i = 0; i < _amounts.length; i++) {
      _totalAmount = _totalAmount.add(_amounts[i]);
    }

    IERC20(xald).safeTransferFrom(msg.sender, address(this),
_totalAmount);
    uint256 _totalShare = IWXALD(wxald).wrap(_totalAmount);

    for (uint256 i = 0; i < _amounts.length; i++) {
      uint256 _share =
_amounts[i].mul(_totalShare).div(_totalAmount);
      shares[_users[i]] = shares[_users[i]].add(_share);
    }
```

Status

The development team has fixed this issue in commit 13dc528.

4.3.4 In the `ALDDaoV2` contract, there is unclear logic for the deposit of the ALD token into the `Staking` contract.

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

Description

The `ALDDaoV2` contract is used to handle the earnings of DAO members, who deposit their DAO tokens under this contract and receive a proportionate share of the earnings. The `redeem()` function is used to withdraw the earnings of the `ALDDaoV2` contract and distribute this to the members. However, no function in the current contract deals with the principal, and it is not clear where the principal of the proceeds comes from when `redeem()` is called.

```
// located in ALDDaoV2.sol
/// @dev redeem xALD reward from staking contract.
function redeem() external {
  IStaking(staking).redeem(address(this), false);
  uint256 _amount =
IWXALD(wxald).wrap(IERC20(xald).balanceOf(address(this)));

  wxALDPerShare =
wxALDPerShare.add(_amount.mul(1e18).div(totalShares));
  }
```

Status

This issue has already been discussed. The development team distributes rewards to the `ALDDaoV2` contract by calling the `stakeFor()` function directly under the `Staking` contract.

4.3.5 Wrong initial condition setting can lead to incorrect calculation results.

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

Description

The `_redeemRewardBondLocks()` function is used to calculate the pending rewards in a vault, which are recorded in separate epochs. The `_lastEpoch` parameter indicates the `epochNumber` of the last reward the user claimed. At the same time, `_shares.length` shows the number of `epochNumber` of pending rewards the user has not yet settled. The code incorrectly compares the two parameters, leading to incorrect results.

```
// located in Staking.sol
function _redeemRewardBondLocks(
    address _user,
    uint256 _lastEpoch,
    uint256 _lastBlock
  ) internal returns (uint256) {

    ......

      for (uint256 _epoch = _lastEpoch; _epoch <
_shares.length; _epoch++) {
        uint256 _share = _shares[_epoch - _lastEpoch];
        if (_share > 0) {
          uint256 _amount;
          uint256 _lockedBlock;
          uint256 _unlockBlock;
          {
            RewardBondBalance storage _lock =
rewardBondLocks[_epoch];
```

```
            uint256 _totalShare =
IRewardBondDepositor(rewardBondDepositor).rewardShares(_epoch,
_vault);
            _amount =
_lock.amounts[_vault].mul(_share).div(_totalShare);
            _lockedBlock = _lock.lockedBlock;
            _unlockBlock = _lock.unlockBlock;
          }
          // [_lockedBlock, _unlockBlock), [_lastBlock + 1,
block.number + 1)
          uint256 _left = Math.max(_lockedBlock, _lastBlock +
1);
          uint256 _right = Math.min(_unlockBlock, block.number
+ 1);
        ......
      }
    }

    return unlockedAmount;
  }
```

Status

The development team has adjusted the calculation strategy to fix this issue in commit a2490d8.

4.3.6 Incorrect use of the `blockNumber` parameter as `epochNumber` parameter will result in incorrect calculation result.

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

Description

The `_findPossibleStartEpoch()` function is used to find the possible start epoch for the current user to calculate pending or unlocked ALD tokens for the vault. The `_lastEpoch` parameter represents the `EpochNumber` of the user's most recent redeem pending reward, but the code incorrectly records it as `blockNumber`, which is inconsistent with code design logic.

```
// located in Staking.sol
function _findPossibleStartEpoch(address _user, uint256
_lastBlock) internal view returns (uint256) {
    uint256 _minLockedBlock =
_findEarlistRewardLockedBlock(_user);
    uint256 _lastEpoch = checkpoint[_user].blockNumber;
    if (_minLockedBlock == 0) {
      // No locks available or all locked ALD are redeemed, in
this case,
      //  + _lastBlock = 0: user didn't interact with the
contract, we should calculate from the first epoch
      //  + _lastBlock != 0: user has interacted with the
contract, we should calculate from the last epoch
      if (_lastBlock == 0) return 0;
      else return _lastEpoch;
    } else {
      // Locks available, we should find the epoch number by
searching _minLockedBlock
      return _findEpochByLockedBlock(_minLockedBlock,
_lastEpoch);
    }
  }
```

Status

The development team has fixed this issue in commit 45bf930.

4.3.7 There is a risk of price manipulation when calling Uniswap V2 to calculate the value of the LP token.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Medium	Flashloan Attack	Fixed

Description

The AladdinDAO (V2) allows users to bond ALD tokens using LP tokens (e.g., ETH-ALD LP token). The Uniswap V2 price data is used to calculate the value of LP tokens, which carries the risk of price manipulation. For example, a user maliciously raises an LP token's value through a flash loan before using the LP token to bond ALD tokens. In this condition, he can get more ALD tokens, which has a serious impact on the stability of the project.

```
// located in DirectBondDepositor.sol
/// @dev deposit token to bond ALD.
/// @param _token The address of token.
/// @param _amount The amount of token.
function deposit(
  address _token,
  uint256 _amount,
  uint256 _minBondAmount
) external nonReentrant {
  require(isBondAsset[_token], "DirectBondDepositor: not
approved");

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

  totalPurchased[_token] =
totalPurchased[_token].add(_amount);

  uint256 _bondAmount;
  if (isLiquidityToken[_token]) {
```

```
    _bondAmount =
ITreasury(treasury).deposit(ITreasury.ReserveType.LIQUIDITY_TO
KEN, _token, _amount);
  } else {
    _bondAmount =
ITreasury(treasury).deposit(ITreasury.ReserveType.UNDERLYING,
_token, _amount);
  }

  require(_bondAmount >= _minBondAmount, "DirectBondDepositor:
bond not enough");

  IStaking(staking).bondFor(msg.sender, _bondAmount);

  emit Deposit(msg.sender, _token, _amount);
}
```

Status

After discussion, the development team used the following three strategies to reduce the risk of price manipulation:

- (1) Only EOA accounts are allowed to call the `deposit()` function;
- (2) A function has been added to the `Staking` contract to allow the administrator to lock out a specific account. If a malicious user is found to be manipulating prices, the administrator will lock the account and not allow the ALD token to be withdrawn;
- (3) Using moving average prices to validate the spot prices obtained in Uniswap V2.

5. Conclusion

After auditing and analyzing the AladdinDAO (V2) contract, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above. SECBIT Labs holds the view that AladdinDAO (V2) contract has good code quality, concise implementation, and detailed documentation.

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 ethers 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