Security Audit Report

Concentrator sdCRV Strategy by AladdinDAO (asdCRV)

SECBIT

June 16, 2023

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 sdCRV revenue strategy to support the deposit of CRV tokens into the Stake DAO protocol. SECBIT Labs conducted an audit from January 9 to January 29, 2023, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the Concentrator sdCRV strategy 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).

Updated on June 16, 2023. SECBIT Labs conducted an review for the ConcentratorVaultForAsdCRV update: add harvester for asdCRV and found no issues.

Type	Description	Level	Status
Design & Implementation	4.3.1 Potential DOS attack can seriously affect the proper functioning logic of the contract.	High	Fixed
Design & Implementation	4.3.2 Discussion of the logic of the function `_checkpoint()`.	Info	Discussed
Gas Optimization	4.3.3 Optimize the function `harvestBribes()` logic to reduce gas consumption.	Info	Discussed
Design & Implementation	4.3.4 The function `claimBribeRewards()` may not be able to claim rewards.	Info	Discussed
Code Revise	4.3.5 Add a restriction on the parameter `_endtime` to improve code clarity.	Info	Discussed
Gas Optimization	4.3.6 Some unused parameters should be removed to save gas.	Info	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the AladdinDAO Concentrator sdCRV strategy is shown below:

- Smart contract code
  - initial review commit *0b7ae48*
  - final review commit *66a014b*
- *Updated on June 16, 2023.* ConcentratorVaultForAsdCRV update.
  - commit *aba2c5a*

2.2 Contract List

The following content shows the contracts included in the AladdinDAO Concentrator sdCRV strategy, which the SECBIT team audits:

Name	Lines	Description
AladdinSdCRV.sol	158	A peripheral contract in which the user deposits CRV tokens and sdVeCRV tokens.
SdCRVLocker.sol	52	An abstract contract inherited by the `AladdinSdCRV.sol` and `StakeDAOCRVvault.sol` contracts, which are used to record information about the withdrawal of funds.
StakeDAOCRVVault.sol	131	It is an underlying vault contract that receives funds from the `AladdinSdCRV.sol` contract and exchanges them into sdCRV tokens. At the same time, it also deals with the user's earnings.
StakeDAOLockerProxy.sol	122	It is a proxy contract that receives the sdCRV tokens from the `StakeDAOCRVVault.sol` contract and deposits them in the Stake DAO gauge.
StakeDAOVaultBase.sol	278	The underlying contract enables the deposit and withdrawal of funds and the distribution of earnings.
VeSDTDelegation.sol	238	A delegated contract to increase user revenue, where users can delegate their veSDT token holdings to this contract to help increase voting power, and it will increase contract revenue.

Updated on June 16, 2023. The ConcentratorVaultForAsdCRV.sol is added for review in this *add harvester for asdCRV* update. It's a Concentrator vault inherited from the ConcentratorGeneralVault contract.

3. Contract Analysis

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

3.1 Role Classification

Two key roles in the AladdinDAO Concentrator sdCRV strategy are Governance Account and Common Account.

- Governance Account
  - Description

    Contract Administrator
  - Authority
    - Update basic parameters
    - 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 Concentrator sdCRV strategy.
  - Authority
    - Deposit CRV tokens or sdVeCRV tokens to receive share tokens (asdCRV tokens)
    - Harvest pending rewards
    - Deposit veSDT token to improve contract yields
  - Method of Authorization

    No authorization required

3.2 Functional Analysis

The sdCRV strategy is part of the Concentrator protocol, which supports depositing CRV tokens into the Stake DAO protocol and receiving the reward. The SECBIT team conducted a detailed audit of some of the contracts in the strategy. We can divide the critical functions of the contract into four parts:

AladdinSdCRV and SdCRVLocker

This contract acts as a peripheral part of the sdCRV strategy, it allows the user to deposit CRV tokens and sdVeCRV tokens, and the user receives the corresponding sdCRV tokens as shares.

The main functions in these contracts are as below:

- `depositWithCRV()`

  This function allows the user to deposit CRV tokens, and the user will receive asdCRV tokens as shares.
- `depositWithsdVeCRV()`

  This function allows the user to deposit sdVeCRV tokens, and the user will receive asdCRV tokens as shares.
- `harvest()`

  Users can call this function to claim the contract rewards and notify it.
- `withdrawExpired()`

  This function allows the user to withdraw expired funds.

StakeDAOCRVVault, StakeDAOVaultBase, and sdCRVLocker

This contract acts as a relay contract. It receives tokens from the `AladdinSdCRV.sol` contract and exchanges them into sdCRV tokens which are then transferred to the `StakeDAOLockerProxy.sol` contract.

The main functions in these contracts are as below:

- `depositWithCRV()`

This function allows the user to deposit CRV tokens, and the user will receive asdCRV tokens as shares.

- `depositWithsdVeCRV()`

  This function allows the user to deposit sdVeCRV tokens, and the user will receive asdCRV tokens as shares.

- `withdraw()`

  The user withdraws sdCRV tokens through this function, and these funds will be locked for a while.

- `harvestBribes()`

  Authorized users receive bribe rewards through this function. If the reward is SDT token, it will be distributed to the users.

StakeDAOLockerProxy

This contract is the main entry for stake tokens in the Stake DAO protocol. The main functions in these contracts are as below:

- `deposit()`

  This function allows the operator to deposit staked tokens to the Stake DAO protocol.

- `withdraw()`

  The operator withdraws staked tokens from the Stake DAO protocol to the recipient.

- `claimRewards()`

  The operator claims pending rewards from the Stake DAO protocol.

- `claimBribeRewards()`

  The operator claims bribe rewards for sdCRV tokens.

VeSDTDelegation

This contract is designed as an auxiliary contract to increase yield by depositing veSDT tokens. The main functions in these contracts are as below:

- `boostPermit()`

  Authorized users can use this function to boost some veSDT tokens to the `StakeDaoLocker Proxy.sol` contract.

- `boost()`

  Users can use this function to boost some veSDT tokens to the `StakeDaoLockerProxy.sol` contract without authorization.

- `claim()`

  Any user can claim SDT tokens rewards for some users by this function.

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 the 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 Potential DOS attack can seriously affect the proper functioning logic of the contract.

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

Description

In `sdCRVLocker.sol`, users who want to retrieve funds deposited in the Aladdin sdCRV strategy must first initiate a fund retrieval request and can only withdraw the funds after the unlocking time has expired. The `withdrawExpired()` function is used to withdraw unlocked funds, where the parameter `_length` indicates the length of the current unlocked funds' array.

This function will retrieve the unlocked funds when the unlocking time arrives. If the `length` is too long for the current unlock time, the function call will fail due to insufficient gas. Three conditions may cause `length` to be too long.

- Suppose a malicious user calls the `withdraw()` function to retrieve funds frequently for a short period. It will generate two simultaneous funds retrieval `locks[_recipient]` records, one for the `StakeDAOCRVVault .sol` contract for the `AladdinSdCRV.sol` contract address to retrieve the sdCRV token and another for the `AladdinSdCRV.sol` contract for the user address to retrieve the sdCRV token. When the user calls the `withdraw()` function at high frequency in a short period, the corresponding `locks[AladdinSdCRV.address]` array will increase rapidly, which will most likely cause the `withdrawExpired()` function call to fail. Since the `locks[AladdinSdCRV.address]` array needs to be generated for all users to retrieve funds, if it is too long, other users will not be able to withdraw funds forever.
- Notice that the functions in the `StakeDAOCRVVault.sol` contract are all of type `external` and have no permission restrictions, which allows the user to call them directly. Suppose a malicious user calls the `depositWithCRV()` / `depositWithsdVeCRV()` functions to deposit funds directly under the contract and set the `_recipient` address to the user's address. Subsequently, the malicious user calls the `withdraw()` function at high frequency and sets the parameter `_recipient` of the `withdraw()` function to the `AladdinSdCRV.sol` contract address, which then also generates a large number of `locks[AladdinSdCRV.address]` array, which also causes the `withdrawExpired()` function call to fail.
- There is also a risk that the `locks[AladdinSdCRV.address]` array will become too long and the `withdrawExpired()` function will fail if a large number of users remove funds from the contract at the same time (or within a short period). In this case, the longer the time to unlock the funds (`_withdrawLockTime` parameter) is set, the higher the corresponding risk.

```
function withdrawExpired(address _user, address _recipient)
external returns (uint256 _amount) {
    if (_user != msg.sender) {
      require(_recipient == _user, "withdraw from others to
others");
    }

    LockedBalance[] storage _locks = locks[_user];
    uint256 _nextIndex = nextLockIndex[_user];
    uint256 _length = _locks.length;

    //@audit it will fail due to insufficient gas if the
length is too large
    while (_nextIndex < _length) {
      LockedBalance memory _lock = _locks[_nextIndex];
      // The list may not be ordered by expireAt, since
`withdrawLockTime` could be changed.
      // However, we will still wait the first one to expire
just for the sake of simplicity.
      if (_lock.expireAt > block.timestamp) break;
      _amount += _lock.amount;

      delete _locks[_nextIndex]; // clear to refund gas
      _nextIndex += 1;
    }
    nextLockIndex[_user] = _nextIndex;

    _unlockToken(_amount, _recipient);

    emit WithdrawExpired(_user, _recipient, _amount);
  }

function _lockToken(uint256 _amount, address _recipient)
internal {
    uint256 _expiredAt = block.timestamp + withdrawLockTime();
    locks[_recipient].push(LockedBalance({ amount:
uint128(_amount), expireAt: uint128(_expiredAt) }));
```

```
    emit Lock(msg.sender, _recipient, _amount, _expiredAt);
  }

//@audit located in AlddinsdCRV .sol
function _withdraw(
    uint256 _shares,
    address _receiver,
    address _owner
  ) internal override returns (uint256) {
    ......
    totalAssetsStored = _totalAssets - _amount; // never
overflow here

    // vault has withdraw fee, we need to subtract from it
    //@audit withdraw funds recorded under this contract
    IStakeDAOCRVVault(vault).withdraw(_amount, address(this));
    uint256 _vaultWithdrawFee =
FeeCustomization(vault).getFeeRate(VAULT_WITHDRAW_FEE_TYPE,
address(this));
    if (_vaultWithdrawFee > 0) {
      _vaultWithdrawFee = (_amount * _vaultWithdrawFee) /
FEE_PRECISION;
      _amount = _amount - _vaultWithdrawFee;
    }

    //@audit record user's funds
    _lockToken(_amount, _receiver);

    emit Withdraw(msg.sender, _receiver, _owner, _amount,
_shares);

    return _amount;
  }

//@audit located in StakeDAOCRVVault.sol
function withdraw(uint256 _amount, address _recipient)
external override(StakeDAOVaultBase, IStakeDAOVault) {
    _checkpoint(msg.sender);
```

```
    ......

    _lockToken(_amount, _recipient);

    emit Withdraw(msg.sender, _recipient, _amount,
_withdrawFee);
  }
```

Status

The development team has modified the logic and has fixed this issue in commit 66a014b.

4.3.2 Discussion of the logic of the function `_checkpoint()`.

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

Description

When processing a user's pending reward via the `_checkpoint()` function, there is an additional determination of whether or not the reward has SDT tokens. If there are no SDT tokens in the current default reward, the code adds extra handling of SDT token rewards. It is necessary to check whether the logic is designed to handle the SDT token that may be claimed in the `harvestBribes()` function.

```
function _checkpoint(address _user) internal override returns
(bool) {
    bool _hasSDT = StakeDAOVaultBase._checkpoint(_user);
    if (!_hasSDT) {
      _checkpoint(SDT, userInfo[_user],
userInfo[_user].balance);
    }
    return true;
```

```
  }

//@audit located in StakeDAOVaultBase.sol
function _checkpoint(address _user) internal virtual returns
(bool _hasSDT) {
    UserInfo storage _userInfo = userInfo[_user];
    uint256 _balance = _userInfo.balance;

    uint256 _count = rewardTokens.length;
    for (uint256 i = 0; i < _count; i++) {
      address _token = rewardTokens[i];
      _checkpoint(_token, _userInfo, _balance);

      if (_token == SDT) _hasSDT = true;
    }
  }

  /// @dev Internal function to update the user information
for specific token.
  /// @param _token The address of token to update.
  /// @param _userInfo The UserInfor struct to update.
  /// @param _balance The total amount of staking token staked
for the user.
  function _checkpoint(
    address _token,
    UserInfo storage _userInfo,
    uint256 _balance
  ) internal {
    RewardData memory _rewardInfo = rewardInfo[_token];
    if (_rewardInfo.periodLength > 0) {
      uint256 _currentTime = _rewardInfo.finishAt;
      if (_currentTime > block.timestamp) {
        _currentTime = block.timestamp;
      }
      uint256 _duration = _currentTime >=
_rewardInfo.lastUpdate ? _currentTime - _rewardInfo.lastUpdate
: 0;
      if (_duration > 0) {
        _rewardInfo.lastUpdate = uint48(block.timestamp);
```

```
            _rewardInfo.accRewardPerShare =
_rewardInfo.accRewardPerShare.add(

 _duration.mul(_rewardInfo.rate).mul(REWARD_PRECISION) /
totalSupply
            );

            rewardInfo[_token] = _rewardInfo;
          }
        }

        // update user information
        if (_balance > 0) {
          _userInfo.rewards[_token] =
uint256(_userInfo.rewards[_token]).add(

 _rewardInfo.accRewardPerShare.sub(_userInfo.rewardPerSharePai
d[_token]).mul(_balance) / REWARD_PRECISION
          );
          _userInfo.rewardPerSharePaid[_token] =
_rewardInfo.accRewardPerShare;
        }
      }
```

Status

Considering the potential SDT Token rewards, the developer team keeps the current logic.

4.3.3 Optimize the function `harvestBribes()` logic to reduce gas consumption.

Risk Type	Risk Level	Impact	Status
Gas Optimization	Info	More gas consumption	Discussed

Description

The user can withdraw the bride rewards under the contract using the `harvestBribes()` function. If the reward token is not SDT token, it will be transferred to the `platform` address, and there is no need to call the `_distribute()` function to distribute the rewards. If the reward token is SDT token, the remaining funds after `_platformFee` and `_boostFee` will be distributed to the user as a reward. Therefore, the `harvestBribes()` function only needs to handle the SDT tokens. The current code does not distinguish between the token type and the reward amount remaining, which increases gas consumption.

```
function harvestBribes(IStakeDAOMultiMerkleStash.claimParam[]
memory _claims) external override {

 IStakeDAOLockerProxy(stakeDAOProxy).claimBribeRewards(_claims
, address(this));

    FeeInfo memory _fee = feeInfo;
    uint256[] memory _amounts = new uint256[](_claims.length);
    address[] memory _tokens = new address[](_claims.length);
    for (uint256 i = 0; i < _claims.length; i++) {
      address _token = _claims[i].token;
      uint256 _reward = _claims[i].amount;
      uint256 _platformFee = uint256(_fee.platformPercentage)
* 100;
      uint256 _boostFee = uint256(_fee.boostPercentage) * 100;

      // Currently, we will only receive SDT as bribe rewards.
      // If there are other tokens, we will transfer all of
them to platform contract.
      if (_token != SDT) {
        _platformFee = FEE_PRECISION;
        _boostFee = 0;
      }
      if (_platformFee > 0) {
        _platformFee = (_reward * _platformFee) /
FEE_PRECISION;
```

```
        IERC20Upgradeable(_token).safeTransfer(_fee.platform,
_platformFee);
      }
      if (_boostFee > 0) {
        _boostFee = (_reward * _boostFee) / FEE_PRECISION;
        IERC20Upgradeable(_token).safeTransfer(delegation,
_boostFee);
      }
      emit HarvestBribe(_token, _reward, _platformFee,
_boostFee);

      _amounts[i] = _reward - _platformFee - _boostFee;
      _tokens[i] = _token;
    }
    _distribute(_tokens, _amounts);
  }
```

Suggestion

The corresponding recommended modifications are as follows.

```
function harvestBribes(IStakeDAOMultiMerkleStash.claimParam[]
memory _claims) external override {
   ......
      // Currently, we will only receiv1e SDT as bribe
rewards.
      // If there are other tokens, we will transfer all of
them to platform contract.
      if (_token != SDT) {
        _platformFee = FEE_PRECISION;
        _boostFee = 0;
      }

      if (_platformFee > 0) {
        _platformFee = (_reward * _platformFee) /
FEE_PRECISION;
        IERC20Upgradeable(_token).safeTransfer(_fee.platform,
_platformFee);
      }
```

```
        if (_boostFee > 0) {
          _boostFee = (_reward * _boostFee) / FEE_PRECISION;
          IERC20Upgradeable(_token).safeTransfer(delegation,
_boostFee);
        }
        emit HarvestBribe(_token, _reward, _platformFee,
_boostFee);

        //@audit modify the following code
        _amounts[i] = _reward - _platformFee - _boostFee;

        if(_token == SDT && _amounts[i] > 0)
          {
            _distribute(_tokens, _amounts[i]);
          }
      }
      //@audit remove the following code
      //_distribute(_tokens, _amounts);
    }
```

Status

Currently, only SDT tokens are awarded through the `harvestBribes()` function, so the development team decided not to change the current code.

4.3.4 The function `claimBribeRewards()` may not be able to claim rewards.

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

Description

The mapping `claimed` is set to prevent users from repeatedly withdrawing rewards, but this may result in normal rewards not being claimed. Consider the following scenario: suppose there are currently two rewards with the same token, but they have different `index`s, so both rewards are valid. After the user has claimed the first reward, the mapping `claimed[_token][_root]` will be set to `true`, so the user will not be able to claim the second token reward.

```
/// @notice The sdCRV bribe claim status for token =>
merkleRoot mapping.
mapping(address => mapping(address => bool)) public claimed;

function
claimBribeRewards(IStakeDAOMultiMerkleStash.claimParam[]
memory _claims, address _recipient)
    external
    override
  {
    require(msg.sender == claimer, "only bribe claimer");
    uint256 _length = _claims.length;
    // 1. claim bribe rewards from StakeDAOMultiMerkleStash
    for (uint256 i = 0; i < _length; i++) {
      // in case someone has claimed the reward for this
contract, we can still call this function to process reward.
      if
(!IStakeDAOMultiMerkleStash(MULTI_MERKLE_STASH).isClaimed(_cla
ims[i].token, _claims[i].index)) {
        IStakeDAOMultiMerkleStash(MULTI_MERKLE_STASH).claim(
          _claims[i].token,
          _claims[i].index,
          address(this),
          _claims[i].amount,
          _claims[i].merkleProof
        );
      }
    }

    // 2. transfer bribe rewards to _recipient
```

```
    for (uint256 i = 0; i < _length; i++) {
      address _token = _claims[i].token;
      address _root =
IStakeDAOMultiMerkleStash(MULTI_MERKLE_STASH).merkleRoot(_toke
n);
      //@audit prevent duplicate awards
      require(!claimed[_token][_root], "bribe rewards
claimed");

      IERC20Upgradeable(_token).safeTransfer(_recipient,
_claims[i].amount);
      claimed[_token][_root] = true;
    }
  }
```

Status

The development team has confirmed that each token corresponds to a unique index, so the above assumption is invalid.

4.3.5 Add a restriction on the parameter `_endtime` to improve code clarity.

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

Description

The function `boost()` is used to increase the revenue obtained from the `stakeDAOProxy` contract. The input parameter `_endtime` indicates when the boost ends, and the parameter `_endtime` must be an integer multiple of 7 and must not exceed four years. It is recommended here to add a judgment on the `_endtime` parameter and give an error message explicitly to improve code clarity.

```
```

```
function boost(
    uint256 _amount,
    uint256 _endtime,
    address _recipient
  ) public {
    require(_recipient != address(0), "recipient is zero
address");
    if (_amount == uint256(-1)) {
      _amount =
IStakeDAOBoostDelegation(veSDT_BOOST).delegable_balance(msg.se
nder);
    }

    IStakeDAOBoostDelegation(veSDT_BOOST).boost(stakeDAOProxy,
_amount, _endtime, msg.sender);

    _boost(_amount, _endtime, _recipient);
  }
```

```
//@audit link:
https://etherscan.io/address/0x47B3262C96BB55A8D2E4F8E3Fed29D2
eAB6dB6e9#code#L183
@internal
def _boost(_from: address, _to: address, _amount: uint256,
_endtime: uint256):
    assert _to not in [_from, ZERO_ADDRESS]
    assert _amount != 0
    assert _endtime > block.timestamp
    //@audit restriction
    assert _endtime % WEEK == 0
    assert _endtime <= VotingEscrow(VE).locked__end(_from)

  ......
```

Suggestion

The corresponding recommended modifications are as follows.

```
function boost(
    uint256 _amount,
    uint256 _endtime,
    address _recipient
  ) public {
    require(_recipient != address(0), "recipient is zero
address");
    //@audit add the following code
    require(_endtime % WEEK == 0, "invalid _endtime");

    if (_amount == uint256(-1)) {
      _amount =
IStakeDAOBoostDelegation(veSDT_BOOST).delegable_balance(msg.se
nder);
    }

    IStakeDAOBoostDelegation(veSDT_BOOST).boost(stakeDAOProxy,
_amount, _endtime, msg.sender);

    _boost(_amount, _endtime, _recipient);
  }
```

Status

This issue has been discussed, and the development team has decided to keep the current logic.

4.3.6 Some unused parameters should be removed to save gas.

Risk Type	Risk Level	Impact	Status
Gas Optimization	Info	More gas consumption	Fixed

Description

The following variables are not used, and it is recommended to remove them or check whether the relevant implementation is complete.

```
/// @dev The address of Stake DAO: SDT Token.
address private constant SDT =
0x73968b9a57c6E53d41345FD57a6E6ae27d6CDB2F; // @audit unused
var

/// @dev The address of Vote-escrowed SDT contract.
// solhint-disable-next-line const-name-snakecase
address private constant veSDT =
0x0C30476f66034E11782938DF8e4384970B6c9e8a; // @audit unused
var

/// @dev The address of veSDT Fee Distributor contract.
  address private constant FEE_DISTRIBUTOR =
0x29f3dd38dB24d3935CF1bf841e6b2B461A3E5D92; // @audit unused
var
```

Status

The development team has removed these parameters in commit 66a014b.

5. Conclusion

After auditing and analyzing the AladdinDAO Concentrator sdCRV strategy 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