Security Audit Report

Concentrator sdCRV Strategy (V2)

by AladdinDAO

SECBIT

December 28, 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. The new version of the protocol has modified the contract functionality by removing the fund withdrawal lock-up period design, allowing users to withdraw their principal at any time. SECBIT Labs conducted an audit from December 15th to December 28th, 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(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 Potential unresolved withdrawal fee discussion.	Info	Discussed
Design & Implementation	4.3.2 When rewards other than SDT tokens are claimed, the emitted event records for `performanceFee` and `boosterFee` values are incorrect.	Info	Discussed
Gas Optimization	4.3.3 When the parameter `_assets` is 0, the remaining code logic can be skipped to save gas.	Info	Fixed
Design & Implementation	4.3.4 Discussion on the reward claiming logic in the `claimRewards()` function.	Info	Discussed
Design & Implementation	4.3.5 Allowing anyone to claim rewards on behalf of a user may result in a loss of user earnings.	Info	Discussed
Design & Implementation	4.3.6 Considering the removal of the function `_lockToken()` to maintain code cleanliness.	Info	Discussed
Design & Implementation	4.3.7 The contract has not set the address for the `DEFAULT_ADMIN_ROLE` role.	Medium	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(V2) is shown below:

- Smart contract code
  - initial review commit *96f3f3a*
  - final review commit *7b5dd64*

2.2 Contract List

The following content shows the contracts included in the AladdinDAO Concentrator sdCRV strategy(v2), which the SECBIT team audits:

Name	Lines	Description
ConcentratorSdCrvGaugeWrapper.sol	83	It is an underlying vault contract that receives funds from the `SdCrvCompounder.sol` contract and exchanges them into sdCRV tokens. At the same time, it also deals with the user's earnings.
ConcentratorStakeDAOGaugeWrapper.sol	187	The underlying contract enables the deposit and withdrawal of funds and the distribution of earnings.
ConcentratorStakeDAOLocker.sol	142	It is a proxy contract that receives the sdCRV tokens from the `ConcentratorSdCrvGaugeWrapper.sol` contract and deposits them in the Stake DAO gauge.
SdCRVBribeBurnerV2.sol	88	A contract that converts other earnings into sdCRV or CRV tokens and distributes them.
SdCrvCompounder.sol	207	A peripheral contract in which the user deposits CRV tokens and sdVeCRV tokens.
LegacyCompounderStash.sol	106	Convert stashed reward tokens to underlying assets and distribute them.
StakeDAOGaugeWrapperStash.sol	23	A reward-claiming contract where protocol earnings are sent to this contract for further processing.

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(v2) 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(v2).
    - Authority
        - Deposit CRV tokens or sdVeCRV tokens or sdCRV-gauge 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 three parts:

SdCrvCompounder & SdCRVLocker

This contract acts as a peripheral part of the sdCRV strategy, it allows the user to deposit CRV tokens, sdVeCRV tokens, or sdCRV-gauge 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.

- `depositWithGauge()`

  This function allows users to transfer sdCRV-gauge tokens to the protocol, and in return, they receive the corresponding asdCRV tokens.

- `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 them.

ConcentratorSdCrvGaugeWrapper & ConcentratorStakeDAOGaugeWrapper

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

- `harvestBribes()`

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

ConcentratorStakeDAOLocker

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.

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 unresolved withdrawal fee discussion.

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

Location

ConcentratorSdCrvGaugeWrapper.sol#L75

Description

Users need to lock their funds for a certain period before they can withdraw their principal using the old version of the `AladdinSdCRV` contract. Considering that funds awaiting withdrawal may be in a lock-up period during a contract upgrade, the `ConcentratorSdCrvGaugeWrapper` contract, during initialization, retrieves funds from users that are awaiting unlocking and transfers them to the upgraded contract. It's important to note that when users use the `AladdinSdCRV.withdraw()` function before the upgrade to reclaim their principal, they also need to pay a certain percentage of _vaultWithdrawFee. This pending fee will be synchronized to the `SdCrvCompounder` contract during the execution of the `initialize()` function. In other words, the parameter `_totalLocked` includes both the user's principal awaiting unlocking and the contract's fees. After the contract upgrade is complete, the potentially existing fees will remain in the `SdCrvCompounder` contract and cannot be processed further.

Based on the provided on-chain contract parameters, it can be confirmed that the current contract has a withdrawal fee rate of 0. Therefore, under these circumstances, utilizing the current code logic won't impact fund allocation.

Overall, if there's no fee charged upon principal withdrawal (i.e. `_vaultWithdrawFee == 0`), then the current code logic is correct. Please ensure that the fee collection aligns with the code logic.

```
function initialize(address _treasury, address _burner) external
initializer {
    __Context_init(); // from ContextUpgradeable
    __ERC165_init(); // from ERC165Upgradeable
    __AccessControl_init(); // from AccessControlUpgradeable
```

```
    __ReentrancyGuard_init(); // from ReentrancyGuardUpgradeable

    __ConcentratorBaseV2_init(_treasury, address(0), _burner); // from
ConcentratorBaseV2
    __LinearMultipleRewardDistributor_init(); // from
LinearMultipleRewardDistributor
    __MultipleRewardAccumulator_init(); // from MultipleRewardAccumulator
    __ConcentratorStakeDAOGaugeWrapper_init(); // from
ConcentratorStakeDAOGaugeWrapper

    if (activeRewardTokens.add(stakingToken)) {
      emit RegisterRewardToken(stakingToken, address(this));
    }

    // sync state from old vault
    address legacyVault = 0x2b3e72f568F96d7209E20C8B8f4F2A363ee1E3F6;
    address asdCRV = 0x43E54C2E7b3e294De3A155785F52AB49d87B9922;
    balanceOf[asdCRV] = IERC20Upgradeable(legacyVault).balanceOf(asdCRV);
    totalSupply = IERC20Upgradeable(legacyVault).totalSupply();
    ISdCRVLocker.LockedBalance[] memory _locks =
ISdCRVLocker(legacyVault).getUserLocks(asdCRV);
    uint256 _totalLocked;
    // @audit according to the logic in the code before the upgrade, the
funds withdrawn here should include the `_vaultWithdrawFee` fee for the
AladdinSdCRV contract.
    for (uint256 i = 0; i < _locks.length; i++) {
      _totalLocked += _locks[i].amount;
    }
    IConcentratorStakeDAOLocker(locker).withdraw(gauge, stakingToken,
_totalLocked, asdCRV);

    // grant role
    _grantRole(DEFAULT_ADMIN_ROLE, _msgSender());

    // approval
    IERC20Upgradeable(CRV).safeApprove(DEPOSITOR, type(uint256).max);
    IERC20Upgradeable(CRV).safeApprove(CURVE_POOL, type(uint256).max);
    IERC20Upgradeable(SD_VE_CRV).safeApprove(DEPOSITOR,
type(uint256).max);
  }
```

Status

The developer has confirmed that the withdrawal fee is 0, and this issue will not affect the actual fund allocation.

4.3.2 When rewards other than SDT tokens are claimed, the emitted event records for `performanceFee` and `boosterFee` values are incorrect.

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

Location

ConcentratorStakeDAOGaugeWrapper.sol#L220

Description

According to the code logic, when the reward claimed is not in SDT token, the contract does not charge `_performanceFee` and `_boosterFee`. However, in this case, the emitted event still records these two values, which is inconsistent with the actual behavior.

```
function harvestBribes(IMultiMerkleStash.claimParam[] memory _claims)
external override nonReentrant {
    IConcentratorStakeDAOLocker(locker).claimBribeRewards(_claims,
address(this));

    address _treasury = treasury;
    address _burner = converter;
    uint256 _expenseRatio = getExpenseRatio();
    uint256 _boosterRatio = getBoosterRatio();
    for (uint256 i = 0; i < _claims.length; i++) {
      address _token = _claims[i].token;
      uint256 _assets = _claims[i].amount;
      uint256 _performanceFee = (_assets * _expenseRatio) /
RATE_PRECISION;
      uint256 _boosterFee = (_assets * _boosterRatio) / RATE_PRECISION;

      // For non-SDT rewards, it will be transfered to BribeBurner
contract waiting for burn.
      // For SDT rewards, it will be distributed intermediately.
      if (_token == SDT) {
        if (_performanceFee > 0) {
```

```
          IERC20Upgradeable(_token).safeTransfer(_treasury,
_performanceFee);
        }
        if (_boosterFee > 0) {
          IERC20Upgradeable(_token).safeTransfer(delegation,
_boosterFee);
        }
        _notifyReward(_token, _assets - _performanceFee - _boosterFee);
      } else {
        IERC20Upgradeable(_token).safeTransfer(_burner, _assets);
      }

      // @audit when token != SDT, the emitted event records incorrect
values
      emit HarvestBribe(_token, _assets, _performanceFee, _boosterFee);
    }
  }
```

Status

The emit here is correct as the fees are still charged based on the current ratio in the `_burner` contract.

4.3.3 When the parameter `_assets` is 0, the remaining code logic can be skipped to save gas.

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

Description

When the parameter `_assets` is 0, there is no need to continue executing the following code logic to save gas.

```
function harvest(address _receiver) external override nonReentrant {
    _checkpoint(address(0));

    address[] memory _tokens = getActiveRewardTokens();
    // claim rewards from locker and stash
    IConcentratorStakeDAOLocker(locker).claimRewards(gauge, new address[]
(0));
```

```
    uint256[] memory _amounts =
StakeDAOGaugeWrapperStash(stash).withdrawTokens(_tokens);

    address _treasury = treasury;
    uint256 _expenseRatio = getExpenseRatio();
    uint256 _harvesterRatio = getHarvesterRatio();
    uint256 _boosterRatio = getBoosterRatio();
    for (uint256 i = 0; i < _tokens.length; i++) {
      address _token = _tokens[i];
      // @audit when _assets is 0, the following code can be skipped.
      uint256 _assets = _amounts[i];
      uint256 _performanceFee;
      uint256 _harvesterBounty;
      uint256 _boosterFee;
      if (_expenseRatio > 0) {
        _performanceFee = (_assets * _expenseRatio) / RATE_PRECISION;
        IERC20Upgradeable(_token).safeTransfer(_treasury,
_performanceFee);
      }
      if (_harvesterRatio > 0) {
        _harvesterBounty = (_assets * _harvesterRatio) / RATE_PRECISION;
        IERC20Upgradeable(_token).safeTransfer(_receiver,
_harvesterBounty);
      }
      if (_tokens[i] == SDT && _boosterRatio > 0) {
        _boosterFee = (_assets * _boosterRatio) / RATE_PRECISION;
        IERC20Upgradeable(_token).safeTransfer(delegation, _boosterFee);
      }

      emit Harvest(_token, _msgSender(), _receiver, _assets,
_performanceFee, _harvesterBounty, _boosterFee);
      unchecked {
        _notifyReward(_token, _assets - _performanceFee -
_harvesterBounty - _boosterFee);
      }
    }
  }
```

Suggestion

The corresponding recommended modifications are as follows.

```
function harvest(address _receiver) external override nonReentrant {
    _checkpoint(address(0));
```

```
    address[] memory _tokens = getActiveRewardTokens();
    // claim rewards from locker and stash
    IConcentratorStakeDAOLocker(locker).claimRewards(gauge, new address[]
(0));
    uint256[] memory _amounts =
StakeDAOGaugeWrapperStash(stash).withdrawTokens(_tokens);

    address _treasury = treasury;
    uint256 _expenseRatio = getExpenseRatio();
    uint256 _harvesterRatio = getHarvesterRatio();
    uint256 _boosterRatio = getBoosterRatio();
    for (uint256 i = 0; i < _tokens.length; i++) {
      address _token = _tokens[i];
      uint256 _assets = _amounts[i];
      //@audit add the following code
      if(_assets == 0) continue;
      uint256 _performanceFee;
      uint256 _harvesterBounty;
      uint256 _boosterFee;
      if (_expenseRatio > 0) {
        _performanceFee = (_assets * _expenseRatio) / RATE_PRECISION;
        IERC20Upgradeable(_token).safeTransfer(_treasury,
_performanceFee);
      }
      if (_harvesterRatio > 0) {
        _harvesterBounty = (_assets * _harvesterRatio) / RATE_PRECISION;
        IERC20Upgradeable(_token).safeTransfer(_receiver,
_harvesterBounty);
      }
      if (_tokens[i] == SDT && _boosterRatio > 0) {
        _boosterFee = (_assets * _boosterRatio) / RATE_PRECISION;
        IERC20Upgradeable(_token).safeTransfer(delegation, _boosterFee);
      }

      emit Harvest(_token, _msgSender(), _receiver, _assets,
_performanceFee, _harvesterBounty, _boosterFee);
      unchecked {
        _notifyReward(_token, _assets - _performanceFee -
_harvesterBounty - _boosterFee);
      }
    }
  }
```

Status

The development team has confirmed this issue and has fixed it in commit 7b5dd64.

4.3.4 Discussion on the reward claiming logic in the `claimRewards()` function.

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

Location

ConcentratorStakeDAOLocker.sol#L152-L161

Description

The `claimRewards()` function can only be invoked through the harvest() function in the `ConcentratorStakeDAOGaugeWrapper` contract. When calling it, the parameters should be set as `locker.claimRewards(gauge, new address[](0));`. The `claimRewards()` function won't execute the code within the two for-loops to calculate the number of tokens rewarded. Based on the current code logic, the administrator can set the address for contract reward delegation using the `updateGaugeRewardReceiver()` function. Subsequently, the rewards of the `ConcentratorStakeDAOLocker` contract will be sent to the delegated address. If the administrator hasn't set an address for contract reward delegation, the rewards will default to being sent to the contract's own address. Assuming the current administrator hasn't set a contract reward delegation address, other users can directly claim rewards from the gauge contract and send them to the `ConcentratorStakeDAOLocker` contract via delegation. In this scenario, this portion of the rewards won't be considered as part of the protocol earnings. Therefore, after deploying the contract, it's crucial to immediately set the reward delegation address for the `ConcentratorStakeDAOLocker` contract and ensure that the delegation address is non-zero to avoid potential income handling issues. Based on the current logic, the two for-loops within the `claimRewards()` function are actually unnecessary and could lead to misunderstandings.

```
function claimRewards(address _gauge, address[] calldata _tokens)
    external
    override
    onlyOperator(_gauge)
    returns (uint256[] memory _amounts)
  {
    uint256 _length = _tokens.length;
    _amounts = new uint256[](_length);
```

```
    // record balances before to make sure only claimed delta tokens will
be transfered.
    for (uint256 i = 0; i < _length; i++) {
      _amounts[i] =
IERC20Upgradeable(_tokens[i]).balanceOf(address(this));
    }
    // This will claim all rewards including SDT.
    ICurveGauge(_gauge).claim_rewards();
    for (uint256 i = 0; i < _length; i++) {
      _amounts[i] =
IERC20Upgradeable(_tokens[i]).balanceOf(address(this)) - _amounts[i];
      if (_amounts[i] > 0) {
        IERC20Upgradeable(_tokens[i]).safeTransfer(msg.sender,
_amounts[i]);
      }
    }
  }

/// @notice Update the reward receiver for the given gauge.
/// @param _gauge The address of gauge to update.
/// @param _newReceiver The address of reward receiver to update.
function updateGaugeRewardReceiver(address _gauge, address _newReceiver)
external onlyOwner {
    address _oldReceiver =
ICurveGauge(_gauge).rewards_receiver(address(this));
    ICurveGauge(_gauge).set_rewards_receiver(_newReceiver);

    emit UpdateGaugeRewardReceiver(_gauge, _oldReceiver, _newReceiver);
}
```

Status

The development team has explicitly stated that they will definitely set up a delegation contract, ensuring that rewards from this contract will not be sent to its own address.

4.3.5 Allowing anyone to claim rewards on behalf of a user may result in a loss of user earnings.

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

Location

SdCrvCompounder.sol#L243-L244

Description

The rewards from the `legacyVault` contract and the `wrapper` contract under this contract can be claimed by third-party users. The claimed rewards will be sent to the `SdCrvCompounder` contract but will not be considered as part of the user's earnings. The funds claimed on behalf of users will be retained within the contract and cannot be processed.

```
function harvest(address _recipient, uint256 _minAssets) external
override nonReentrant returns (uint256 assets) {
    ensureCallerIsHarvester();

    _distributePendingReward();

    uint256 _amountCRV;
    uint256 _amountSdCRV;
    uint256 _amountETH;
    address _zap = zap;
    // 1.1 claim pending rewards
    {
      // We are prettier sure that all tokens are active
      address[] memory _tokens =
IWrapper_SdCrvCompounder(wrapper).getActiveRewardTokens();
      uint256[] memory _balances = new uint256[](_tokens.length);
      for (uint256 i = 0; i < _balances.length; ++i) {
        _balances[i] =
IERC20Upgradeable(_tokens[i]).balanceOf(address(this));
      }
      // some rewards are still in legacy vault
      // @audit rewards can be claimed by others on behalf of a user
      IWrapper_SdCrvCompounder(legacyVault).claim(address(this),
address(this));
      IWrapper_SdCrvCompounder(wrapper).claim(address(this),
address(this));


      for (uint256 i = 0; i < _balances.length; i++) {
        address _token = _tokens[i];
        uint256 _amount =
IERC20Upgradeable(_tokens[i]).balanceOf(address(this)) - _balances[i];
        if (_token == CRV) {
```

```
            _amountCRV += _amount;
          } else if (_token == sdCRV) {
            _amountSdCRV += _amount;
          } else {
            // convert to ETH
            IERC20Upgradeable(_token).safeTransfer(_zap, _amount);
            _amountETH += IZap(zap).zap(_token, _amount, address(0), 0);
          }
        }
      }
      ......
    }
```

Status

After the upgrade, the `legacyVault` contract no longer distributes new rewards. Once the remaining rewards are claimed, the contract becomes inactive. The `wrapper` contract is configured with a reward delegation contract. Therefore, if a user claims rewards on behalf of others, those rewards will be sent to the delegated address without affecting the reward distribution.

4.3.6 Considering the removal of the function `_lockToken()` to maintain code cleanliness.

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

Location

SdCRVLocker.sol#L86-L99

Description

After the contract upgrade, the logic requiring the locking of sdCRV tokens has been removed. Therefore, it is confirmed that the `_lockToken()` function will no longer be used. It is recommended to remove the `_lockToken()` function to maintain code cleanliness.

```
/// @dev Internal function to lock staking token.
/// @param _amount The amount of staking token to lock.
/// @param _recipient The address of recipient who will receive the
locked token.
function _lockToken(uint256 _amount, address _recipient) internal {
  uint256 _expiredAt = block.timestamp + withdrawLockTime();
```

```
    // ceil up to 86400 seconds
    _expiredAt = ((_expiredAt + DAYS - 1) / DAYS) * DAYS;

    uint256 _length = locks[_recipient].length;
    if (_length == 0 || locks[_recipient][_length - 1].expireAt !=
_expiredAt) {
      locks[_recipient].push(LockedBalance({ amount: uint128(_amount),
expireAt: uint128(_expiredAt) }));
    } else {
      locks[_recipient][_length - 1].amount += uint128(_amount);
    }

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

Status

This issue has been discussed.

4.3.7 The contract has not set the address for the `DEFAULT_ADMIN_ROLE` role.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Medium	Functional issue	Fixed

Location

LegacyCompounderStash.sol#L84-L90

Description

The `LegacyCompounderStash` contract inherits the `AccessControl` contract but forgets to set the default administrator `DEFAULT_ADMIN_ROLE`. This oversight directly results in the inability to set the subsequent `HARVESTER_ROLE`.

```
 contract LegacyCompounderStash is AccessControl {
   constructor(address _compounder) {
      address _asset = ILegacyCompounder(_compounder).asset();
      IERC20(_asset).safeApprove(_compounder, type(uint256).max);

      compounder = _compounder;
      asset = _asset;
    }
...
}
```

Suggestion

Add the setting for `DEFAULT_ADMIN_ROLE` in the constructor as follows:

```
constructor(address _compounder) {
      address _asset = ILegacyCompounder(_compounder).asset();
      IERC20(_asset).safeApprove(_compounder, type(uint256).max);

      compounder = _compounder;
      asset = _asset;

      // @auditz add following code
      _grantRole(DEFAULT_ADMIN_ROLE, _msgSender());
    }
```

Status

The development team has fixed it in commit 7b5dd64.

5. Conclusion

After auditing and analyzing the AladdinDAO Concentrator sdCRV strategy(v2) 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