Security Audit Report

Concentrator Harvester by AladdinDAO

SECBIT

March 8, 2023

1. Introduction

The AladdinDAO network is designed to democratize crypto investment decisions by empowering the collective intelligence of the community. The Concentrator offers a yield enhancement solution that leverages Convex LP assets to farm high-quality DeFi tokens (CRV, CVX, 3CRV). SECBIT Labs conducted an audit for the Concentrator Harvester code from February 23 to March 8, 2023, which focused on identifying code bugs, logic flaws, and potential security risks. The audited contract is based on an older version of the Concentrator, with the `harvest()` function now restricted to users who meet specific conditions. The assessment results indicate that the Concentrator Harvester has no critical security risks. Nevertheless, the SECBIT team has provided tips on logical implementation, potential risks, and code revision in Part 4 of the audit report.

Type	Description	Level	Status
Design & Implementation	4.3.1 The implementation inside the function `hasPermission()` is incorrect.	Medium	Fixed
Design & Implementation	4.3.2 The function `harvesterStorage()` reads the storage slot position incorrectly.	Medium	Fixed
Implementation	4.3.3 The code implementation does not match the requirements document.	Low	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the Concentrator Harvester Facet contract is shown below:

- Project website
  - https://concentrator.aladdin.club/
- Smart contract code
  - https://github.com/AladdinDAO/aladdin-v3-contracts
  - initial review commit *d1128f7*
  - final review commit *3eb75e9*

2.2 Contract List

The following content shows the contracts included in the Harvester Facet of Concentrator, which the SECBIT team audits:

Name	Lines	Description
CLeverAMOHarvesterFacet.sol	9	Harvest pending rewards from the `CLeverAMO` contract.
ConcentratorHarvesterFacet.sol	52	Harvest pending rewards from the `Concentrator` vault and `AladdinCompounder` contract.
StakeDaoHarvesterFacet.sol	19	Harvest pending rewards from `StakeDAOVault` and corresponding `AladdinCompounder` contract.
LibConcentratorHarvester.sol	60	The library contract that verifies user permissions.
ConcentratorBase.sol	24	The contract that verifies user addresses.

In addition to the contracts listed in the table above, we also reviewed the following contracts during the audit.

AladdinCompounder.sol, AladdinCompounderWithStrategy.sol, ConcentratorConvexVault.sol, ConcentratorGeneralVault.sol, CLeverAMOBase.sol, AladdinCVX.sol, AladdinCRV.sol, AladdinCRVConvexVault.sol, AladdinCRVV2.sol, AladdinFXS.sol, and AladdinSdCRV.sol.

These contracts have undergone minor adjustments based on the old version. We have confirmed that these changes do not pose any security risks.

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 Concentrator Harvester Facet: Governance Account and Harvester Account.

- Governance Account
  - Description

    Contract administrator
  - Authority
    - Update white list and black list
    - Update permission
    - Transfer ownership
  - Method of Authorization

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

    The executor of the `harvest()` function
  - Authority
    - Harvest pending rewards from relevant strategies
  - Method of Authorization

    It is necessary to obtain authorization from the administrator or hold enough veCTR tokens.

3.2 Functional Analysis

The Concentrator allows Convex liquidity providers to stake their LP tokens and get diversified benefits. The current code has modified the mode for harvest earnings. The old version of the code allowed anyone to harvest the rewards of the strategy, while the current version requires users to meet certain conditions before they harvest earnings.

The SECBIT team conducted a detailed audit of some of the contracts in the protocol. Here are the crucial functions within the contract:

CLeverAMOHarvesterFacet & ConcentratorHarvesterFacet & StakeDaoHarvesterFacet

These three contracts add calling restrictions to the `harvest()` function for all strategies in the concentrator. The main functions in these contracts are as below:

- `harvestCLeverAMO()`

  Harvest pending rewards from the `CLeverAMO` contract.
- `harvestConcentratorVault()`

  Harvest pending rewards from the Concentrator vault.
- `harvestConcentratorCompounder()`

  Harvest pending rewards from the `AladdinCompounder` contract.
- `harvestStakeDaoVault()`

  Harvest pending rewards from the `StakeDAOVault` contract.
- `harvestStakeDaoVaultAndCompounder()`

  Harvest pending rewards from StakeDAOVault and the corresponding `AladdinCompounder` contract.

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 followed SECBIT Lab's audit specification and involved an analysis of the project's code for bugs, logical implementation, and potential risks. The process consisted of four steps:

- A line-by-line analysis of the contract code
- Evaluation of vulnerabilities and potential risks in the contract code
- Communication on assessment and confirmation
- Audit report writing

4.2 Audit Result

After using open-source tools, including Mythril, Slither, SmartCheck, and Securify, and scanning with SECBIT Labs' internal tools, including adelaide, sf-checker, and badmsg.sender, the auditing team performed a manual assessment of the code. The results were categorized as follows:

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 The implementation inside the function `hasPermission()` is incorrect.

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

Description

The `hasPermission()` function is used to determine whether the specified `_account` address has permission to perform the harvest operation. However, the implementation inside the function is incorrect.

In particular, the internal function `locked()` is used to read the veCTR token information locked by the specified address. However, the `msg.sender` parameter was used instead of `_account`, which means that the function reads the veCTR token information locked under the caller's address, rather than the fund locking information of the specified `_account` address.

To fix this issue, we recommend replacing `msg.sender` with `_account` as the parameter for the `locked()` function. This will ensure that the function reads the correct veCTR token information and returns the correct permission status for the specified `_account` address.

```
function hasPermission(address _account) external view returns
(bool) {
    // @audit the veCTR token locking information
    //        of the `_account` address should be read
    ICurveVoteEscrow.LockedBalance memory _locked =
ICurveVoteEscrow(LibConcentratorHarvester.veCTR).locked(msg.se
nder);
    LibConcentratorHarvester.HarvesterStorage storage hs =
LibConcentratorHarvester.harvesterStorage();

    return
      hs.whitelist[_account] ||
      (uint128(_locked.amount) >= hs.minLockCTR && _locked.end
>= hs.minLockDuration + block.timestamp);
  }
```

Suggestion

The code has been modified to the following:

```
function hasPermission(address _account) external view returns
(bool) {
    // @audit use the _account parameter instead of
msg.sender.
    ICurveVoteEscrow.LockedBalance memory _locked =
ICurveVoteEscrow(LibConcentratorHarvester.veCTR).locked(_accou
nt);

    LibConcentratorHarvester.HarvesterStorage storage hs =
LibConcentratorHarvester.harvesterStorage();

    return
      hs.whitelist[_account] ||
      (uint128(_locked.amount) >= hs.minLockCTR && _locked.end
>= hs.minLockDuration + block.timestamp);
  }
```

Status

The team fixed this issue in commit 3eb75e9.

4.3.2 The function `harvesterStorage()` reads the storage slot position incorrectly.

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

Description

The `harvesterStorage()` function is used to return the `HarvesterStorage` struct, which stores various state variables related to the harvester. However, the function reads the storage slot position of the `DiamondStorage` struct, which may cause confusion and result in incorrect data storage.

To fix this issue, we recommend updating the function to use the correct storage slot position for the `HarvesterStorage` struct. This will ensure that the function returns the correct state variables for the harvester and prevents any potential confusion or incorrect data storage.

It's important to note that this issue may potentially cause unexpected behavior in the contract, such as incorrect state variables being returned or overwritten.

```
/// @dev The storage slot for default diamond storage.
bytes32 private constant DIAMOND_STORAGE_POSITION =
keccak256("diamond.standard.diamond.storage");

/// @dev The storage slot for harvester storage.
bytes32 private constant HARVESTER_STORAGE_POSITION =
keccak256("diamond.harvester.concentrator.storage");

struct DiamondStorage {
```

```
    // function selector => facet address and selector
position in selectors array
    mapping(bytes4 => FacetAddressAndSelectorPosition)
facetAddressAndSelectorPosition;
    bytes4[] selectors;
    mapping(bytes4 => bool) supportedInterfaces;
    // owner of the contract
    address contractOwner;
  }

struct HarvesterStorage {
    uint128 minLockCTR;
    uint128 minLockDuration;
    mapping(address => bool) whitelist;
  }

function diamondStorage() private pure returns (DiamondStorage
storage ds) {
    bytes32 position = DIAMOND_STORAGE_POSITION;
    assembly {
      ds.slot := position
    }
  }

  function harvesterStorage() internal pure returns
(HarvesterStorage storage hs) {
    //@audit the correct storage slot position
    //       to read is `HARVESTER_STORAGE_POSITION`
    bytes32 position = DIAMOND_STORAGE_POSITION;
    assembly {
      hs.slot := position
    }
  }
```

Status

The team fixed this issue in commit 3eb75e9.

4.3.3 The code implementation does not match the requirements document.

Risk Type	Risk Level	Impact	Status
Implementation	Low	Incomplete logic	Fixed

Description

We have identified a discrepancy between the code implementation and the requirements document. Specifically, the current code logic does not meet the condition in the document that states: "If the veCTR token amount is set to zero, the lock time of the veCTR token should not be checked."

It's important to confirm with the development team whether this requirement is still valid and needs to be implemented. If the requirement is still valid, we recommend updating the code to ensure that the lock time of the veCTR token is not checked if the veCTR token amount is set to zero.

```
function enforceHasPermission() internal view {
    ICurveVoteEscrow.LockedBalance memory _locked =
ICurveVoteEscrow(veCTR).locked(msg.sender);
    HarvesterStorage storage hs = harvesterStorage();

    // check whether is whitelisted
    if (hs.whitelist[msg.sender]) return;

    // check veCTR locking
    require(uint128(_locked.amount) >= hs.minLockCTR,
"insufficient lock amount");
    require(_locked.end >= hs.minLockDuration +
block.timestamp, "insufficient lock duration");
  }
```

Suggestion

According to the requirements in the document, the code should been modified as follows:

```
function enforceHasPermission() internal view {
    ICurveVoteEscrow.LockedBalance memory _locked =
ICurveVoteEscrow(veCTR).locked(msg.sender);
    HarvesterStorage storage hs = harvesterStorage();

    // check whether is whitelisted
    // @audit the code has been modified to the following
    if (hs.whitelist[msg.sender] || hs.minLockCTR == 0)
return;

    // check veCTR locking
    require(uint128(_locked.amount) >= hs.minLockCTR,
"insufficient lock amount");
    require(_locked.end >= hs.minLockDuration +
block.timestamp, "insufficient lock duration");
  }
```

Status

The team fixed this issue in commit 3eb75e9.

5. Conclusion

After auditing and analyzing the Harvester Facet of Concentrator protocol, 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