Security Audit Report

aFXS by AladdinDAO

SECBIT

August 17, 2022

1. Introduction

The AladdinDAO is a decentralized network to shift crypto investments from venture capitalists to the wisdom of crowds through collective value discovery. The new aFXS protocol extends the use case for Aladdin products by adding a new revenue strategy that allows users to deposit cvxFXSFXS-f tokens for ongoing revenue. SECBIT Labs conducted an audit from August 3 to August 17, 2022, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the Concentrator contract has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising(see part 4 for details).

Type	Description	Level	Status
Design & Implementation	4.3.1 The potential arithmetic underflow could drain the assets in the pool.	Medium	Fixed
Design & Implementation	4.3.2 Confirmation of withdrawing fee logic during the withdrawal of assets by the user.	Info	Discussed
Design & Implementation	4.3.3 Confusing code design logic.	Medium	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the aFXS contract is shown below:

- Smart contract code
  - initial review commit *9bce6fb*
  - final review commit *4170ac2*

2.2 Contract List

The following content shows the contracts included in Aladdin aFXS protocol, which the SECBIT team audits:

Name	Lines	Description
AladdinCompounder.sol	219	Abstract contract that implements the core logic of the cvxFXSFXS-f token.
ConcentratorConvexVault.sol	435	Abstract contract for depositing LP tokens into Convex Protocol.
AladdinFXS.sol	151	It inherits the `AladdinCompounder` contract and implements specific functions such as deposit, withdrawal, and reward distribution.
AladdinFXSConvexVault.sol	94	It inherits the `ConcentratorConvexVault` contract and allows users to choose the form they receive their rewards.

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 Aladdin aFXS protocol: Governance Account and Common Account.

- Governance Account
    - Description

      Contract Administrator
    - Authority
        - Update basic parameters

        - Update the percentage of various fees charged
        - 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 Aladdin aFXS protocol.
    - Authority
        - Deposit cvxFXSFXS-f token and claim rewards
        - Harvest gains from Convex and distribute it
    - Method of Authorization

      No authorization required

3.2 Functional Analysis

The aFXS expands the revenue strategy for Aladdin products. The new contract allows users to deposit cvxFXSFXS-f token, which will be transferred to the Convex protocol to earn interest. 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 two parts:

AladdinCompounder & AladdinFXS

This contract is an abstract contract, which, together with the AladdinFXS contract, forms a complete system. Users deposit cvxFXSFXS-f tokens into this contract to receive a share of aFXS tokens. And as time goes on, the aFXS tokens held by the user will be redeemed for more cvxFXSFXS-f tokens.

The main functions in `AladdinCompounder` and `AladdinFXS` are as below:

- `deposit()`

This function allows the user to deposit the cvxFXSFXS-f tokens in this contract, and these tokens will be transferred to the Convex protocol simultaneously.

- `mint()`

  The code will calculate the amount of cvxFXSFXS-f tokens that the user should deposit based on the number of aFXS tokens the user wants to obtain, and it will transfer these cvxFXSFXS-f tokens to the Convex protocol.

- `withdraw()`

  The user withdraws a specified number of cvxFXSFXS-f tokens assets and burns the corresponding share of aFXS tokens.

- `redeem()`

  The user specifies the amount of aFXS tokens to be burned and gets back the corresponding cvxFXSFXS-f tokens.

- `harvest()`

  Users can call this function to harvest rewards from the Convex protocol.

AladdinFXSConvexVault & ConcentratorConvexVault

With the two contracts mentioned above, users can deposit the cvxFXSFXS-f token directly into the convex protocol and receive rewards. Different from the `AladdinCompounder` contract, the user of this contract can choose the type of rewards he wants to receive.

The main functions in `AladdinFXSConvexVault` and `ConcentratorConvexVault` are as below:

- `deposit()`

  The user deposits funds into the specified strategy, which will be recorded under the `_recipient` address.

- `withdraw()`

The user withdraws funds from the specified strategy to the specified `_recipient` address.

- `claim()`

  This function allows the user to withdraw the earnings that have been generated under the specified strategy.

- `claimAll()`

  This function allows the user to withdraw earnings already generated under all strategies.

- `harvest()`

  Users can call this function to retrieve revenue from the Convex protocol and distribute it to all users.

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 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 potential arithmetic underflow could drain the assets in the pool.

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

Description

Firstly, the function `harvest()` is used to retrieve earnings in Convex and release them linearly as rewards to AladdinFXS protocol users. The internal function `_notifyHarvestedReward()` is used to update the number of rewards released per unit of time. At the same time, this function updates the parameter `_info.lastUpdate` and the parameter `_info.finishAt`.

```
// @audit located in AladdinFXS.sol
function harvest(address _recipient, uint256 _minAssets)
external override nonReentrant returns (uint256) {
    _distributePendingReward();

    ......

    // 3. update rewards info
    _depositToConvex(_amountLP);
    _notifyHarvestedReward(_amountLP - _platformFee -
_harvestBounty);

    return _amountLP;
  }

// @audit located in AladdinCompounder.sol
```

```
function _notifyHarvestedReward(uint256 _amount) internal
virtual {
    RewardInfo memory _info = rewardInfo;
    if (_info.periodLength == 0) {
      totalAssetsStored = totalAssetsStored.add(_amount);
    } else {
      require(_amount < uint128(-1), "amount overflow");

      if (block.timestamp >= _info.finishAt) {
        _info.rate = uint128(_amount / _info.periodLength);
      } else {
        uint256 _remaining = _info.finishAt - block.timestamp;
        uint256 _leftover = _remaining * _info.rate;
        _info.rate = uint128((_amount + _leftover) /
_info.periodLength);
      }

      _info.lastUpdate = uint48(block.timestamp);
      _info.finishAt = uint48(block.timestamp +
_info.periodLength);

      rewardInfo = _info;
    }
  }
```

Secondly, the internal function `_notifyHarvestedReward()` is used to distribute the currently released rewards. It is called by the `deposit()`, `mint()`, `withdraw()` and `redeem()` functions. Note that this internal function first updates the number of rewards released at the current time and then updates the time parameter `rewardInfo.lastUpdate`, which may introduce a security risk.

Consider the following scenario: No user may call the `harvest()` function during the `periodLength` period, in which case there is `block.timestamp > _info.finishAt`. Given the above context, assume that the attacker calls the `deposit()` function to deposit a large number of cvxFXSFXS-f tokens into the contract and obtains the corresponding aFXS token. At this point, the `deposit()` function calls the `_distributePendingReward()` and updates

the `lastUpdate` parameter to the current timestamp. The attacker immediately calls the `withdraw()` function to get back the specified amount of principal. Now, the `withdraw()` function still calls the `_distributePending Reward()` function. Note that the condition `block.timestamp > _info.finishAt` still holds, but the value of the `lastUpdate` parameter has been updated to `block.timestamp`, so the following two conditions hold.

```
rewardInfo.lastUpdate = uint48(block.timestamp);

block.timestamp > _info.finishAt;
```

Then it follows that

```
rewardInfo.lastUpdate > _info.finishAt;
```

In this case, the code `_info.finishAt - _info.lastUpdate` underflows. It directly results in a huge value for `totalAssetsStored`. Theoretically, an attacker could exchange a tiny number of aFXS tokens for a huge number of cvxFXSFXS-f tokens.

```
function deposit(uint256 _assets, address _receiver) public
override nonReentrant returns (uint256) {
   ......

    _distributePendingReward();

    ......
  }


 function mint(uint256 _shares, address _receiver) external
override nonReentrant returns (uint256) {
    _distributePendingReward();

    ......
  }
```

```
function withdraw(
    uint256 _assets,
    address _receiver,
    address _owner
  ) external override nonReentrant returns (uint256) {
    _distributePendingReward();

    ......
  }


  function redeem(
    uint256 _shares,
    address _receiver,
    address _owner
  ) public override nonReentrant returns (uint256) {
    if (_shares == uint256(-1)) {
      _shares = balanceOf(_owner);
    }
    _distributePendingReward();

   ......
  }


  function _distributePendingReward() internal virtual {
    RewardInfo memory _info = rewardInfo;
    if (_info.periodLength == 0) return;

    uint256 _period;

    if (block.timestamp > _info.finishAt) {

      //@audit risk code
      _period = _info.finishAt - _info.lastUpdate;
    } else {
      _period = block.timestamp - _info.lastUpdate;
    }
```

```
        ......

        rewardInfo.lastUpdate = uint48(block.timestamp);
    }
```

In addition, there is a risk that the `totalAssets()` function called inside the `withdraw()` function will also underflow in the above case.

```
function withdraw(
    uint256 _assets,
    address _receiver,
    address _owner
  ) external override nonReentrant returns (uint256) {
    _distributePendingReward();

    uint256 _totalAssets = totalAssets();
    ......
  }

function totalAssets() public view virtual override returns
(uint256) {
    RewardInfo memory _info = rewardInfo;
    uint256 _period;
    if (block.timestamp > _info.finishAt) {

      //@audit risk code
      _period = _info.finishAt - _info.lastUpdate;
    } else {
      _period = block.timestamp - _info.lastUpdate;
    }
    return totalAssetsStored + _period * _info.rate;
  }
```

Status

The team confirmed this issue and fixed it in commit 980f944.

4.3.2 Confirmation of withdrawing fee logic during the withdrawal of assets by the user.

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

Description

The `withdraw()` function retrieves the principle that a user has deposited into the aFXS protocol, and the user will also receive corresponding revenue. When calculating the actual amount of funds available to the user, the function also calculates the amount of `_withdrawFee` that the user should pay, but the code does not handle this further, so this design logic needs to be confirmed.

```
//@auidt located in AladdinFXS.sol
function _withdraw(
    uint256 _shares,
    address _receiver,
    address _owner
  ) internal override returns (uint256) {
    require(_shares > 0, "aFXS: withdraw zero share");
    require(_shares <= balanceOf(_owner), "aFXS: insufficient
owner shares");
    uint256 _totalAssets = totalAssetsStored; // the value is
correct
    uint256 _totalShare = totalSupply();
    uint256 _amount = _shares.mul(_totalAssets) / _totalShare;
    _burn(_owner, _shares);

    if (_totalShare != _shares) {
      // take withdraw fee if it is not the last user.
      uint256 _withdrawFee = (_amount *
feeInfo.withdrawPercentage) / FEE_DENOMINATOR;

      //@audit codes to be confirmed
      _amount = _amount - _withdrawFee; // never overflow here
```

```
    } else {
      // @note If it is the last user, some extra rewards
still pending.
      // We just ignore it for now.
    }

    totalAssetsStored = _totalAssets - _amount; // never
overflow here

    _withdrawFromConvex(_amount, _receiver);

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

    return _amount;
  }
```

Status

This issue has been discussed. No need to change.

4.3.3 Confusing code design logic.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Medium	More gas consumption	Fixed

Description

The function `pendingReward()` calculates the user's unclaimed rewards. The parameter `_lastUpdate` is assigned the current timestamp when the following conditions are satisfied.

```
_lastUpdate > block.timestamp
```

At this point, we have

```
_lastUpdate = block.timestamp;

_rewardInfo.lastUpdate > block.timestamp;
```

As a result, the calculation of `_lastUpdate - _rewardInfo.lastUpdate` will underflow, directly affecting the calculation of the user's unclaimed rewards.

```
//@audit located in ConcentratorConvexVault.sol
function pendingReward(uint256 _pid, address _account) public
view override returns (uint256) {
    PoolInfo storage _pool = poolInfo[_pid];
    RewardInfo memory _rewardInfo = rewardInfo[_pid];

    uint256 _accRewardPerShare = _pool.accRewardPerShare;
    if (_rewardInfo.periodLength > 0) {
      uint256 _lastUpdate = _rewardInfo.lastUpdate;

      // solhint-disable-next-line not-rely-on-time
      // @audit risk code
      if (_lastUpdate > block.timestamp) _lastUpdate =
block.timestamp;
      uint256 _duration = _lastUpdate -
_rewardInfo.lastUpdate;

      if (_duration > 0 && _pool.totalShare > 0) {
        _accRewardPerShare =
_accRewardPerShare.add(_duration.mul(_rewardInfo.rate).mul(PRE
CISION) / _pool.totalShare);
      }
    }

    return _pendingReward(_pid, _account, _accRewardPerShare);
  }
```

The same situation exists for the `_updateRewards()` function.

```
function _updateRewards(uint256 _pid, address _account)
internal virtual {
    PoolInfo storage _pool = poolInfo[_pid];

    // 1. update global info
    RewardInfo memory _rewardInfo = rewardInfo[_pid];
    uint256 _accRewardPerShare = _pool.accRewardPerShare;
    if (_rewardInfo.periodLength > 0) {
      uint256 _lastUpdate = _rewardInfo.lastUpdate;
      // solhint-disable-next-line not-rely-on-time
      if (_lastUpdate > block.timestamp) _lastUpdate =
block.timestamp;
      uint256 _duration = _lastUpdate -
_rewardInfo.lastUpdate;
      ......
  }
```

Status

The team confirmed this issue and modified the logic in commit 980f944.

5. Conclusion

After auditing and analyzing the Aladdin aFXS 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