Security Audit Report

Fx Rebalance Pool With Boost

by AladdinDAO

SECBIT

December 13, 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. As a part of the AladdinDAO ecosystem, the f(x) protocol creates two new ETH derivative assets, one with stablecoin-like low volatility called fractional ETH (fETH) and the second a leveraged long ETH perpetual token called leveraged ETH (xETH). As part of the foundational infrastructure for the f(x) protocol, the rebalance pool is a farming vault for fETH which earns high yields (in stETH) sourced from the staking yields of the reserve. The new version of the rebalance pool introduces a liquidity mining incentive mechanism for FXN tokens, further enhancing user returns. SECBIT Labs conducted an audit from November 29 to December 13, 2023, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the dynamic allocation mechanism for FXN tokens 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 range set for the xToken leverage threshold does not match the actual obtained leverage ratio values.	Medium	Fixed
Gas Optimization	4.3.2 Remove redundant timestamp operations under the update user address to save gas.	Info	Fixed
Design & Implementation	4.3.3 The parameter usage in the function `_checkpoint(address(this))` is not reasonable.	Info	Fixed
Design & Implementation	4.3.4 Discussing the proposal for calculating users' FXN token rewards using veFXN token weights.	Info	Fixed
Gas Optimization	4.3.5 Add a check for the parameter `fullEarned` to save gas.	Info	Fixed
Design & Implementation	4.3.6 The calculation of users' FXN token earnings may be incorrect.	Medium	Fixed
Design & Implementation	4.3.7 There is a potential for a flash loan attack that could drain rewards from the protocol.	Info	Discussed
Gas Optimization	4.3.8 The return value `BoostCheckpoint` in the function `_getBoostRatioRead()` is redundant and could result in a significant waste of gas.	Info	Fixed
Design & Implementation	4.3.9 A malicious user could falsely increase the stored funds in the contract to boost the overall yield for all users.	Info	Discussed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the dynamic allocation mechanism of FXN tokens in the f(x) protocol is shown below:

- Smart contract code
  - initial review commit
    - b33fed8
  - final review commit
    - 097ead8

2.2 Contract List

The following content shows the contracts included in the dynamic allocation mechanism of FXN tokens in the f(x) protocol, which the SECBIT team audits:

Name	Lines	Description
BoostableRebalancePool.sol	405	The new version of the rebalance pool introduces a liquidity mining incentive mechanism for FXN tokens, further enhancing user returns.
RebalancePoolGaugeClaimer.sol	132	The contract for dynamically adjusting the allocation of FXN tokens.
MultipleRewardCompoundingAccumulator.sol	160	The contract responsible for users claiming earnings is a component of the `BoostableRebalancePool.sol` 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 dynamic allocation mechanism of FXN tokens in the f(x) protocol are Governance Account and Common Account.

- Governance Account
  - Description

    Contract Administrator
  - Authority
    - Update protocol parameter
    - Transfer ownership
  - Method of Authorization

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

    Deposit fTokens to earn returns
  - Authority
    - Deposit / Withdraw fTokens
    - Claim profit
    - Claim pending FXN from gauge and split to rebalance pools.
  - Method of Authorization

    No authorization required

3.2 Functional Analysis

The f(x) protocol implements a decentralized quasi-stablecoin with high collateral utilization efficiency and leveraged contracts with low liquidation risks and no funding costs. It uses a Rebalance Pool as its primary mechanism to liquidate collateralized debt positions that fall below the minimum collateralization ratio. The `BoostableRebalancePool.sol` contract, an upgraded version of the rebalance pool, introduces a dynamic allocation mechanism for FXN token rewards. Users can receive up to 2.5 times the FXN token earnings based on the weight of their veFXN token holdings. 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:

BoostableRebalancePool & MultipleRewardCompoundingAccumulator

Stability deposits absorb and cancel the debt resulting from defaulted markets: an equivalent amount of shares in the Rebalance Pool, corresponding to the liquidated treasury's debt, is burned from the pool's balance to offset the debt. As a reward, Rebalance Pool depositors acquire collateral (in stETH) from the liquidated positions at a considerable discount. The proportion of a stability depositor's current deposit to the total fETH in the pool determines the collateral share they receive from the liquidation process.

The main functions in this contract are as follows:

- `deposit()`

  Anyone could call this function to deposit some asset (fETH) to this contract.

- `withdraw()`

  Users can withdraw the fTokens they have deposited into the contract at any time, as long as these fTokens have not been liquidated.

- `claim()`

  Users can claim pending rewards from this contract.

- `liquidate()`

  When the collateralization ratio is too low, users can use this function to perform liquidation, which helps to increase the system's collateralization ratio.

RebalancePoolGaugeClaimer

Dynamic allocation contract for FXN tokens. The main functions in this contract are as follows:

- `claim()`

  Claim pending FXN from the gauge and split to rebalance pools.

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 The range set for the xToken leverage threshold does not match the actual obtained leverage ratio values.

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

Location

RebalancePoolGaugeClaimer.sol#L207

Description

The precision of the leverage ratio obtained from the `treasury.leverageRatio()` function for xToken is 1e18, while the precision for setting the leverage ratio threshold of xToken in the current contract is 1e9. In this scenario, the leverage ratio `_leverageRatio` obtained by the contract significantly exceeds the contract's set upper limit (default value is 3e9, with a maximum configurable value of 5e9). The `if` condition branch is always true, and the parameter `_splitterRatio` constantly equals `minSplitterRatio` (default value is 666,666,666), which is inconsistent with the actual requirements.

```
/// @dev The minimum value of `leveratio_raio_min`.
uint256 private constant MIN_LEVERAGE_RATIO_LOWER_BOUND = 13e8; // 1.3

/// @dev The maximum value of `leveratio_raio_min`.
uint256 private constant MAX_LEVERAGE_RATIO_LOWER_BOUND = 2e9; // 2

/// @dev The minimum value of `leveratio_raio_max`.
uint256 private constant MIN_LEVERAGE_RATIO_UPPER_BOUND = 2e9; // 2

/// @dev The maximum value of `leveratio_raio_max`.
uint256 private constant MAX_LEVERAGE_RATIO_UPPER_BOUND = 5e9; // 5


function _computeSplitterRatio() internal view returns (uint256
_splitterRatio) {
    SplitterRatioParameters memory _params = params;

    // @audit the _leverageRatio precision is 1e18
    uint256 _leverageRatio = IFxTreasury(treasury).leverageRatio();

    //@audit if branch always evaluates to true.
    if (_leverageRatio > _params.leverageRatioUpperBound) {
      _splitterRatio = _params.minSplitterRatio;
    } else if (_leverageRatio < _params.leverageRatioLowerBound) {
      _splitterRatio = PRECISION;
    } else {
      // a = (leverageRatioLowerBound * minSplitterRatio -
leverageRatioUpperBound) / c
      // b = (1 - minSplitterRatio) / c
      // c = leverageRatioLowerBound - leverageRatioUpperBound
      // a + b * leverageRatio
      //   = leverageRatioLowerBound * minSplitterRatio -
leverageRatioUpperBound + (1 - minSplitterRatio) * leverageRatio
      //   = minSplitterRatio * (leverageRatioLowerBound - leverageRatio)
+ leverageRatio - leverageRatioUpperBound
      unchecked {
        _splitterRatio =
          (uint256(_params.minSplitterRatio) *
            (_leverageRatio - uint256(_params.leverageRatioLowerBound)) +
            (uint256(_params.leverageRatioUpperBound) - _leverageRatio) *
            PRECISION) /
          uint256(_params.leverageRatioUpperBound -
_params.leverageRatioLowerBound);
      }
    }
  }
```

Status

The development team has confirmed the issue and addressed it by modifying the contract's leverage precision to 1e18 in the commit 097ead8.

4.3.2 Remove redundant timestamp operations under the update user address to save gas.

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

Location

BoostableRebalancePool.sol#L248

BoostableRebalancePool.sol#L276

Description

Within the `_checkpoint()` function, the user parameter `_balance.updateAt` has already been updated; therefore, there is no need to redundantly update this parameter in the subsequent code. Doing so would result in unnecessary gas consumption.

```
  function deposit(uint256 _amount, address _receiver) external override {
      address _sender = _msgSender();
      // transfer asset token to this contract
      address _asset = asset;
      if (_amount == type(uint256).max) {
        _amount = IERC20Upgradeable(_asset).balanceOf(_sender);
      }
      if (_amount == 0) revert DepositZeroAmount();
      IERC20Upgradeable(_asset).safeTransferFrom(_sender, address(this),
  _amount);

      // @note after checkpoint, the account balances are correct, we can
  `_balances` safely.
      _checkpoint(_receiver);
```

```
    // It should never exceed `type(uint104).max`.
    TokenBalance memory _supply = _totalSupply;
    TokenBalance memory _balance = _balances[_receiver];
    _supply.amount += uint104(_amount);
    _supply.updateAt = uint40(block.timestamp);
    _balance.amount += uint104(_amount);

    //@audit redundant operation
    _balance.updateAt = uint40(block.timestamp);

    _recordTotalSupply(_supply);
    _totalSupply = _supply;
    _balances[_receiver] = _balance;

    emit Deposit(_sender, _receiver, _amount);
    emit UserDepositChange(_receiver, _balance.amount, 0);
  }

function withdraw(uint256 _amount, address _receiver) external override {
    address _sender = _msgSender();

    // @note after checkpoint, the account balances are correct, we can
`_balances` safely.
    _checkpoint(_sender);

    TokenBalance memory _supply = _totalSupply;
    TokenBalance memory _balance = _balances[_sender];
    if (_amount > _balance.amount) {
      _amount = _balance.amount;
    }
    if (_amount == 0) revert WithdrawZeroAmount();

    unchecked {
      _supply.amount -= uint104(_amount);
      _supply.updateAt = uint40(block.timestamp);
      _balance.amount -= uint104(_amount);

      // @audit redundant operation
      _balance.updateAt = uint40(block.timestamp);
    }

    _recordTotalSupply(_supply);
    _balances[_sender] = _balance;
    _totalSupply = _supply;
```

```
        IERC20Upgradeable(asset).safeTransfer(_receiver, _amount);

        emit Withdraw(_sender, _receiver, _amount);
        emit UserDepositChange(_sender, _balance.amount, 0);
      }

    function _checkpoint(address _account) internal virtual override {
        ......
        TokenBalance memory _balance = _balances[_account];
        TokenBalance memory _supply = _totalSupply;

        uint104 _newBalance = uint104(_getCompoundedBalance(_balance.amount,
    _balance.product, _supply.product));
        if (_newBalance != _balance.amount) {
          // no unchecked here, just in case
          emit UserDepositChange(_account, _newBalance, _balance.amount -
    _newBalance);
          }

        _balance.amount = _newBalance;
        _balance.product = _supply.product;

        // @audit update _account's timestamp
        _balance.updateAt = uint40(block.timestamp);
        _balances[_account] = _balance;
      }
    }
```

Status

The development team has adopted our suggestions and removed the redundant code in commit 097ead8.

4.3.3 The parameter usage in the function `_checkpoint(address(this))` is not reasonable.

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

Location

BoostableRebalancePool.sol#L296

Description

The `_checkpoint()` function initially claims the FXN tokens obtained from the contract's mining activities, followed by updating the earnings and capital changes for the specified `_account` (which may involve liquidation) and the user's principal fTokens. Using `_checkpoint(address(this))` with the contract address as a user seems unnecessary since, under normal circumstances, the contract address wouldn't hold any fToken principal.

If a user accidentally transfers fTokens to the contract, those funds should be considered permanently lost and not require further processing. Changing `address(this)` to `address(0)` appears to align more closely with the code logic. In the case of the zero address, only the FXN tokens obtained from the contract's mining activities would be claimed. It is advisable to verify whether the code `_checkpoint(address(this))` meets the desired requirements.

```
function liquidate(uint256 _maxAmount, uint256 _minBaseOut)
    external
    override
    onlyRole(LIQUIDATOR_ROLE)
    returns (uint256 _liquidated, uint256 _baseOut)
  {
    //@audit use address(this) is confusing
    _checkpoint(address(this));

    IFxTreasury _treasury = IFxTreasury(treasury);
    if (_treasury.collateralRatio() >= liquidatableCollateralRatio) {
      revert CannotLiquidate();
    }
    (, uint256 _maxLiquidatable) =
_treasury.maxRedeemableFToken(liquidatableCollateralRatio);

    uint256 _amount = _maxLiquidatable;
    if (_amount > _maxAmount) {
      _amount = _maxAmount;
    }

    address _asset = asset;
    address _market = market;
    address _wrapper = wrapper;
    _liquidated = IERC20Upgradeable(_asset).balanceOf(address(this));
    if (_amount > _liquidated) {
      // cannot liquidate more than assets in this contract.
      _amount = _liquidated;
    }
    IERC20Upgradeable(_asset).safeApprove(_market, 0);
    IERC20Upgradeable(_asset).safeApprove(_market, _amount);
    (_baseOut, ) = IFxMarket(_market).redeem(_amount, 0, _wrapper,
_minBaseOut);
    _liquidated = _liquidated -
IERC20Upgradeable(_asset).balanceOf(address(this));

    emit Liquidate(_liquidated, _baseOut);

    // wrap base token if needed
    address _token = baseToken;
    if (_wrapper != address(this)) {
      _baseOut = IFxTokenWrapper(_wrapper).wrap(_baseOut);
      _token = IFxTokenWrapper(_wrapper).dst();
    }

    // distribute liquidated base token
    _accumulateReward(_token, _baseOut);

    // notify loss
    _notifyLoss(_liquidated);
  }
```

Status

The development team has acknowledged the issue and addressed it by modifying the code to `_checkpoint(address(0))` in commit 097ead8.

4.3.4 Discussing the proposal for calculating users' FXN token rewards using veFXN token weights.

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

Location

BoostableRebalancePool.sol#389

BoostableRebalancePool.sol#560

Description

When calculating the quantity of FXN tokens a user receives, the code takes into account the user's holdings of veFXN tokens. It's important to note that when using the veFXN token weight of a user, the code only reads the veFXN token weight from the last time the user deposited/withdrew funds and the current veFXN token weight the user holds. The code estimates the veToken weight changes linearly over time during this period. The actual changes in a user's veFXN weight may deviate from this linear estimation, introducing potential risks.

```
function _checkpoint(address _account) internal virtual override {
  // fetch FXN from gauge every 24h
  Gauge memory _gauge = gauge;
  if (_gauge.gauge != address(0) && block.timestamp >
uint256(_gauge.claimAt) + DAY) {
    uint256 _balance = IERC20Upgradeable(fxn).balanceOf(address(this));
    ICurveTokenMinter(minter).mint(_gauge.gauge);
    uint256 _minted = IERC20Upgradeable(fxn).balanceOf(address(this)) -
_balance;
    gauge.claimAt = uint64(block.timestamp);
    _notifyReward(fxn, _minted);
  }

  MultipleRewardCompoundingAccumulator._checkpoint(_account);

  if (_account != address(0)) {
    boostCheckpoint[_account] = BoostCheckpoint(
      // @audit read the weight of veFXN and snap it
      uint112(IVotingEscrowProxy(veProxy).adjustedVeBalance(_account)),
      uint112(IVotingEscrow(ve).totalSupply()),
      uint32(numTotalSupplyHistory)
    );


     ......
}

function _getBoostRatioRead(address _account) internal view returns
(uint256, BoostCheckpoint memory) {
    TokenBalance memory _balance = _balances[_account];
    BoostCheckpoint memory _boostCheckpoint = boostCheckpoint[_account];
    // no deposit before
    if (_balance.amount == 0) return (0, _boostCheckpoint);

    // @note For gas saving, we assume the ve balance and ve supply are
changing linearly.
    // @audit weight of veFXN token currently
    uint256 _currentVeBalance =
IVotingEscrowProxy(veProxy).adjustedVeBalance(_account);
    uint256 _currentVeSupply = IVotingEscrow(ve).totalSupply();

    (uint256 _currentRatio, uint256 _nextIndex) = _boostRatioAt(
      _balance,
      _boostCheckpoint.veBalance,
      _boostCheckpoint.veSupply,
      _boostCheckpoint.historyIndex,
      _balance.updateAt
    );
    if (uint256(_balance.updateAt) == block.timestamp) {
      _boostCheckpoint.veBalance = uint112(_currentVeBalance);
      _boostCheckpoint.veSupply = uint112(_currentVeSupply);
      _boostCheckpoint.historyIndex = uint32(_nextIndex);
      return (_currentRatio, _boostCheckpoint);
    }

    int256 _deltaVeBalance = int256(_currentVeBalance) -
int256(uint256(_boostCheckpoint.veBalance));
    int256 _deltaVeSupply = int256(_currentVeSupply) -
int256(uint256(_boostCheckpoint.veSupply));
    int256 _duration = int256(block.timestamp - _balance.updateAt);

    uint256 _prevTs = _balance.updateAt;
    uint256 _accumulatedRatio;
    // compute the time weighted boost from _balance.updateAt to now.
    uint256 _nowTs = ((_prevTs + WEEK - 1) / WEEK) * WEEK;
    for (uint256 i = 0; i < 256; ++i) {
      // it is more than 4 years, should be enough
      if (_nowTs > block.timestamp) _nowTs = block.timestamp;
      _accumulatedRatio += _currentRatio * (_nowTs - _prevTs);
      if (_nowTs == block.timestamp) break;
      uint256 _veBalance;
      uint256 _veSupply;
      {
        int256 dt = int256(_nowTs - _balance.updateAt);
        // @audit estimate veFXN token balance
        _veBalance = uint256((_deltaVeBalance * dt) / _duration +
int256(uint256(_boostCheckpoint.veBalance)));
        _veSupply = uint256((_deltaVeSupply * dt) / _duration +
int256(uint256(_boostCheckpoint.veSupply)));
      }
      (_currentRatio, _nextIndex) = _boostRatioAt(_balance, _veBalance,
_veSupply, _nextIndex, _nowTs);
      _prevTs = _nowTs;
      _nowTs += WEEK;
    }

    _boostCheckpoint.veBalance = uint112(_currentVeBalance);
    _boostCheckpoint.veSupply = uint112(_currentVeSupply);
    _boostCheckpoint.historyIndex = uint32(_nextIndex);
    return (_accumulatedRatio / uint256(_duration), _boostCheckpoint);
  }
```

The contract employs the `IVotingEscrowProxy(veProxy).adjustedVeBalance(_account)` interface to retrieve the veFXN token weight held by a user. When `veBoost` is the zero address, it directly obtains the weight of the veFXN tokens. When `veBoost` is not the zero address, it also allows delegating the weight held by others to this user. Assuming Alice deposits a certain amount of fTokens in the first week and withdraws this portion of funds by the 10th week without any intervening actions, consider the following two scenarios:

(1). The contract does not support a delegation mechanism to increase the user's veFXN token weight. For instance, Alice can lock FXN tokens for only one week before depositing fTokens. Subsequently, the user calls the deposit function to invest fToken funds. After one week, Alice's lock on the FXN tokens expires, she can withdraw these FXN tokens. By the 10th week, Alice locks FXN tokens again for one week and then calls the `withdraw()` function to retrieve fTokens. When calculating the earnings for Alice, the code calculates based on the assumption that Alice's FXN tokens were locked for ten weeks. However, in reality, Alice only locked FXN tokens in the 1st and 10th weeks and had control over the FXN tokens during the intervening time.

(2). The contract supports a delegation mechanism to increase the user's veFXN token weight. Assuming Bob currently holds 100 veFXN tokens, before Alice deposits fTokens, Bob can delegate his 100 veFXN tokens to Alice, with the delegation time set to one week. As time goes on, Bob can delegate the veFXN tokens to Charlie again, and so on. Before Alice wants to claim the earnings, Bob can delegate his veFXN tokens to Alice once again. Bob's 100 veFXN tokens will be repeatedly considered in the boost ratio calculation for different users.

In both scenarios described above, Alice can achieve the effect of locking FXN tokens throughout the entire process by only locking FXN tokens before depositing fToken funds and claiming earnings.

```
// @audit https://github.com/AladdinDAO/aladdin-v3-
contracts/blob/b33fed8b9ae5ea983c59436510b8685f242c762c/contracts/voting-
escrow/VotingEscrowProxy.sol#L39-L46
function adjustedVeBalance(address _account) external view override
returns (uint256) {
    address _veBoost = veBoost;
    if (_veBoost == address(0)) {
      return IVotingEscrow(ve).balanceOf(_account);
    } else {
      // @audit read from VotingEscrowBoost
      return IVotingEscrowBoost(_veBoost).balanceOf(_account);
    }
  }
```

Status

To address this issue, the development team has implemented two measures: First, users are not allowed to change the veFXN token weight through delegation. This prevents malicious users from repeatedly influencing the veFXN token weight through delegation for arbitrage. Second, users are allowed to increase FXN token earnings by locking FXN before depositing or claiming funds. This approach will become the default rule for the protocol, and all protocol users will be informed. The corresponding code adjustments can be found in commit 097ead8.

4.3.5 Add a check for the parameter `fullEarned` to save gas.

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

Location

BoostableRebalancePool.sol#416

Description

When the parameter `fullEarned` is equal to 0, such as when a user makes an initial deposit of fTokens, it is possible to skip the subsequent series of unnecessary operations to save the user's gas.

```
function _updateSnapshot(address _account, address _token) internal
virtual override {
    UserRewardSnapshot memory _snapshot = userRewardSnapshot[_account]
[_token];
    uint48 epochExponent = _totalSupply.product.epochAndExponent();

    if (_token == fxn) {
      // @audit skipped the following code if fullEarned is 0
      uint256 fullEarned = _claimable(_account, _token) -
_snapshot.rewards.pending;
      uint256 ratio = _getBoostRatioWrite(_account);
      uint256 boostEarned = (fullEarned * ratio) / PRECISION;
      _snapshot.rewards.pending += uint128(boostEarned);
      if (fullEarned > boostEarned) {
        // redistribute unboosted rewards.
        _notifyReward(fxn, fullEarned - boostEarned);
      }
    } else {
      _snapshot.rewards.pending = uint128(_claimable(_account, _token));
    }
    _snapshot.checkpoint = epochToExponentToRewardSnapshot[_token]
[epochExponent];
    _snapshot.checkpoint.timestamp = uint64(block.timestamp);
    userRewardSnapshot[_account][_token] = _snapshot;
  }
```

Suggestion

The proposed modification is as follows:

```
function _updateSnapshot(address _account, address _token) internal
virtual override {
    UserRewardSnapshot memory _snapshot = userRewardSnapshot[_account]
[_token];
    uint48 epochExponent = _totalSupply.product.epochAndExponent();

    if (_token == fxn) {

      uint256 fullEarned = _claimable(_account, _token) -
_snapshot.rewards.pending;
      //@audit add this check
      if(fullEarned > 0){
       uint256 ratio = _getBoostRatioWrite(_account);
      uint256 boostEarned = (fullEarned * ratio) / PRECISION;
      _snapshot.rewards.pending += uint128(boostEarned);
```

```
        if (fullEarned > boostEarned) {
          // redistribute unboosted rewards.
          _notifyReward(fxn, fullEarned - boostEarned);
          }
        }
      } else {
        _snapshot.rewards.pending = uint128(_claimable(_account, _token));
      }
      _snapshot.checkpoint = epochToExponentToRewardSnapshot[_token]
[epochExponent];
      _snapshot.checkpoint.timestamp = uint64(block.timestamp);
      userRewardSnapshot[_account][_token] = _snapshot;
    }
```

Status

The development team has accepted our suggestions and updated the code logic in commit 097ead8.

4.3.6 The calculation of users' FXN token earnings may be incorrect.

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

Location

BoostableRebalancePool.sol#618

Description

The mapping `totalSupplyHistory` stores daily records of the total funds under the protocol. These records are used to calculate the user's `boostRatio`, which determines the amount of FXN tokens the user can receive. The parameter `numTotalSupplyHistory` represents the current number of historical records in the mapping `totalSupplyHistory`, and this value increases over time in the `_recordTotalSupply()` function.

In normal circumstances, when a user deposits or withdraws funds, their `boostCheckpoint` snapshot should reflect the latest `numTotalSupplyHistory` value for the current day. However, the first user to call the `deposit()` or `withdraw()` function each day still uses the `numTotalSupplyHistory` value from the previous day. In other words, at the beginning of a new day, when the first user calls the `deposit()` or `withdraw()` function, they initially record the `numTotalSupplyHistory` value from the previous day in their `boostCheckpoint` snapshot before updating the `numTotalSupplyHistory` value.

Let's detail the scenario where the first user calls the `deposit()` function to deposit funds at the beginning of a new day. Assume the protocol is currently running on the 2nd day, and no users have called the `deposit()`, `withdraw()`, or `liquidate()` functions yet. At this point, the parameter `numTotalSupplyHistory` is set to 1. When the first user calls the `deposit()` function, they will initially execute the `_checkpoint(_receiver)` function. Inside the `_checkpoint(_receiver)` function, a snapshot is taken of the user's veFXN weight information and the day number, which is `numTotalSupplyHistory = 1`. Subsequently, the contract calls the `_recordTotalSupply()` function, updating the `numTotalSupplyHistory` value to 2. After this point, users who deposit funds, withdraw funds, or participate in liquidation on the 2nd day will use the snapshot with `numTotalSupplyHistory = 2`.

The difference in snapshot states between the first user of each day and other users can have implications, particularly in scenarios where the parameter `numTotalSupplyHistory` is used, as seen in the function `_boostRatioAt()`.

Considering the assumed scenario:

- Alice, the first user of the new day, calls the `deposit()` function.
- An hour later, Bob, another user, also calls the `deposit()` function (mainly to trigger the update of the `_supply.updateAt` parameter).
- Subsequently, Alice calls the `claim()` function to receive FXN token rewards.

When Alice calls the `claim()` function, it triggers the `_checkpoint()` function, followed by a call to the `_boostRatioAt()` function to calculate the `boostRatio` value. Referring to the comments in the `_boostRatioAt()` function, it's evident that there is an error in the calculation of the `_boostedBalance` parameter, further affecting the `boostRatio` value. Consequently, users may receive an incorrect amount of FXN rewards.

```
/// @inheritdoc IFxBoostableRebalancePool
function deposit(uint256 _amount, address _receiver) external override {
    ......

    // @note after checkpoint, the account balances are correct, we can
`_balances` safely.
    _checkpoint(_receiver);

    ......

    _recordTotalSupply(_supply);
    _totalSupply = _supply;
    _balances[_receiver] = _balance;
    emit Deposit(_sender, _receiver, _amount);
    emit UserDepositChange(_receiver, _balance.amount, 0);
  }

 function _checkpoint(address _account) internal virtual override {
    ......

    MultipleRewardCompoundingAccumulator._checkpoint(_account);

    if (_account != address(0)) {
      boostCheckpoint[_account] = BoostCheckpoint(
        uint112(IVotingEscrowProxy(veProxy).adjustedVeBalance(_account)),
        uint112(IVotingEscrow(ve).totalSupply()),
        uint32(numTotalSupplyHistory)
      );

      TokenBalance memory _balance = _balances[_account];
      TokenBalance memory _supply = _totalSupply;

      ......
  }

function _recordTotalSupply(TokenBalance memory _supply) internal {
    unchecked {
      uint256 _numTotalSupplyHistory = numTotalSupplyHistory;
      TokenBalance memory _last =
totalSupplyHistory[_numTotalSupplyHistory - 1];
      if (_last.updateAt / DAY == _supply.updateAt / DAY) {
        totalSupplyHistory[_numTotalSupplyHistory - 1] = _supply;
      } else {
        totalSupplyHistory[_numTotalSupplyHistory] = _supply;
        //@audit Update the value of numTotalSupplyHistory
        numTotalSupplyHistory = _numTotalSupplyHistory + 1;
      }
    }
  }

  function _boostRatioAt(
    TokenBalance memory _balance,
    uint256 _veBalance,
    uint256 _veSupply,
    uint256 startIndex,  //@audit Under the assumed conditions described
above, set this value to 1.
    uint256 t            //@audit The timestamp at which Alice calls the
deposit function.
```

```
  ) internal view returns (uint256, uint256) {
    // @note since totalSupplyHistory is bounded by the number of days,
    // it should be ok to use while loop.
    unchecked {
      //@audit Under the assumed conditions described above, set this
value to 2.
      uint256 endIndex = numTotalSupplyHistory;
      //@audit Start the loop
      while (startIndex < endIndex) {
       //@audit Due to Bob's deposit operation, the value of
totalSupplyHistory[startIndex].updateAt is set to Bob's timestamp. At
this point, the if condition is satisfied, and the loop is terminated.
        if (totalSupplyHistory[startIndex].updateAt > t) break;
        startIndex += 1;
      }

      // @audit startIndex is set to 0
      if (startIndex > 0) startIndex -= 1;
    }
    // @audit Please note that at this moment, the protocol reads the
fund and liquidation status for the first day, but Alice has not yet
deposited any funds.
    TokenBalance memory _supply = totalSupplyHistory[startIndex];

    // @audit Due to the incorrect use of the _supply value, the
calculation results here may be inaccurate.
    uint256 _realBalance = _getCompoundedBalance(_balance.amount,
_balance.product, _supply.product);
    if (_realBalance == 0) {
      return ((PRECISION * 4) / 10, startIndex);
    }
    uint256 _boostedBalance = (_realBalance * 4) / 10;
    if (_veSupply > 0) {

      //@audit The _supply.amount value used here is also incorrect.
      _boostedBalance += (((_veBalance * uint256(_supply.amount)) /
_veSupply) * 6) / 10;
    }
    if (_boostedBalance > _realBalance) {
      _boostedBalance = _realBalance;
    }

    return ((_boostedBalance * PRECISION) / _realBalance, startIndex);
  }
```

Status

The development team has modified the update condition for the parameter `numTotalSupplyHistory`. Now, depositing or withdrawing funds at different block heights will trigger the update of this parameter, instead of being limited to once per day. The corresponding code changes can be found in commit 097ead8.

4.3.7 There is a potential for a flash loan attack that could drain rewards from the protocol.

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

Location

BoostableRebalancePool.sol#411

Description

The protocol allows anyone to help users claim FXN token earnings through the `claim()` function.

```
// @audit MultipleRewardCompoundingAccumulator.sol
function claim(address _account, address _receiver) public override
nonReentrant {
    if (_account != _msgSender() && _receiver != address(0)) {
      revert ClaimOthersRewardToAnother();
    }

    _checkpoint(_account);
    _claim(_account, _receiver);
  }
```

If the `BoostRatio` value for a user's address does not reach 100%, there will be remaining FXN tokens distributed as rewards to all users. If the reward distribution scheme chooses to distribute rewards all at once, there may be a flash loan attack risk. Consider the following scenario:

Firstly, an attacker utilizes a flash loan to borrow a large amount of fTokens from the market and deposits them into the protocol through the `deposit()` function. Since the protocol allows the attacker to claim rewards on behalf of any user, if the `BoostRatio` value for the user being claimed for is below 100%, the remaining FXN tokens are distributed as rewards.

In this situation, the attacker "helps" numerous legitimate users claim FXN token rewards, potentially resulting in a massive one-time distribution of FXN token rewards to all users.

In this scenario, because the attacker holds the majority of the fTokens deposited in the contract, they can call the `claim()` function to receive a substantial amount of FXN token rewards.

```
function _updateSnapshot(address _account, address _token) internal
virtual override {
    UserRewardSnapshot memory _snapshot = userRewardSnapshot[_account]
[_token];
    uint48 epochExponent = _totalSupply.product.epochAndExponent();

    if (_token == fxn) {
      uint256 fullEarned = _claimable(_account, _token) -
_snapshot.rewards.pending;

      // @audit the `ratio` value may fall below 100%,
      uint256 ratio = _getBoostRatioWrite(_account);

      uint256 boostEarned = (fullEarned * ratio) / PRECISION;
      _snapshot.rewards.pending += uint128(boostEarned);
      if (fullEarned > boostEarned) {
        // redistribute unboosted rewards.
        // @audit the remaining FXN tokens will be distributed as rewards
        _notifyReward(fxn, fullEarned - boostEarned);
      }
    } else {
      _snapshot.rewards.pending = uint128(_claimable(_account, _token));
    }
    _snapshot.checkpoint = epochToExponentToRewardSnapshot[_token]
[epochExponent];
    _snapshot.checkpoint.timestamp = uint64(block.timestamp);
    userRewardSnapshot[_account][_token] = _snapshot;
  }

 // @audit LinearMultipleRewardDistributor.sol
 function _notifyReward(address _token, uint256 _amount) internal {
    // @audit the administrator has the option to distribute rewards all
at once.
    if (periodLength == 0) {
      _accumulateReward(_token, _amount);
    } else {
      LinearReward.RewardData memory _data = rewardData[_token];
```

```
        _data.increase(periodLength, _amount);
        rewardData[_token] = _data;
      }
    }
```

Status

The development team has confirmed that the current protocol does not allow for the one-time distribution of FXN token rewards. During protocol deployment, the administrator will set a linear release time for each round of FXN token rewards, and this time cannot be modified by anyone, including the administrator.

4.3.8 The return value `BoostCheckpoint` in the function `_getBoostRatioRead()` is redundant and could result in a significant waste of gas.

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

Location

BoostableRebalancePool.sol#553

Description

It's noteworthy that the function `_getBoostRatioRead()` returns the `boostCheckpoint` struct and is used to update the user's `boostCheckpoint[_account]` value in the function `_getBoostRatioWrite()`. This redundancy seems unnecessary since `_getBoostRatioWrite()` is an internal function of the `_checkpoint()` function, and the `_checkpoint()` function subsequently updates the user's `boostCheckpoint[_account]` value again, leading to duplicate updates of the parameter.

It's important to note that the `boostCheckpoint[_account].historyIndex` updated in the `_getBoostRatioWrite()` function may also be incorrect (`historyIndex` might be incorrect), but this is likely corrected by subsequent updates.

```
  function _checkpoint(address _account) internal virtual override {
      // fetch FXN from gauge every 24h
      Gauge memory _gauge = gauge;
      if (_gauge.gauge != address(0) && block.timestamp >
  uint256(_gauge.claimAt) + DAY) {
        uint256 _balance = IERC20Upgradeable(fxn).balanceOf(address(this));
      ICurveTokenMinter(minter).mint(_gauge.gauge);
      uint256 _minted = IERC20Upgradeable(fxn).balanceOf(address(this)) -
_balance;
      gauge.claimAt = uint64(block.timestamp);
      _notifyReward(fxn, _minted);
    }

    // @audit call _updateSnapshot() function to update
boostCheckpoint[_account]
    MultipleRewardCompoundingAccumulator._checkpoint(_account);


    // @audit update boostCheckpoint[_account] again
    if (_account != address(0)) {
      boostCheckpoint[_account] = BoostCheckpoint(
        uint112(IVotingEscrowProxy(veProxy).adjustedVeBalance(_account)),
        uint112(IVotingEscrow(ve).totalSupply()),
        uint32(numTotalSupplyHistory)
      );

      TokenBalance memory _balance = _balances[_account];
      TokenBalance memory _supply = _totalSupply;

      uint104 _newBalance =
uint104(_getCompoundedBalance(_balance.amount, _balance.product,
_supply.product));
      if (_newBalance != _balance.amount) {
        // no unchecked here, just in case
        emit UserDepositChange(_account, _newBalance, _balance.amount -
_newBalance);
      }

      _balance.amount = _newBalance;
      _balance.product = _supply.product;
      _balance.updateAt = uint40(block.timestamp);
      _balances[_account] = _balance;
    }
  }


function _getBoostRatioWrite(address _account) internal returns (uint256)
{
    (uint256 _ratio, BoostCheckpoint memory _newCheckpoint) =
_getBoostRatioRead(_account);
```

```
    // @audit the recording here appears to be unnecessary, as the user's
`boostCheckpoint` will be updated again in the outermost `_checkpoint()`
function.
    boostCheckpoint[_account] = _newCheckpoint;
    return _ratio;
  }

// @audit there is no need to return the `BoostCheckpoint` struct.
function _getBoostRatioRead(address _account) internal view returns
(uint256, BoostCheckpoint memory) {
    TokenBalance memory _balance = _balances[_account];
    BoostCheckpoint memory _boostCheckpoint = boostCheckpoint[_account];
    // no deposit before
    if (_balance.amount == 0) return (0, _boostCheckpoint);

    // @note For gas saving, we assume the ve balance and ve supply are
changing linearly.
    uint256 _currentVeBalance =
IVotingEscrowProxy(veProxy).adjustedVeBalance(_account);
    uint256 _currentVeSupply = IVotingEscrow(ve).totalSupply();

    (uint256 _currentRatio, uint256 _nextIndex) = _boostRatioAt(
      _balance,
      _boostCheckpoint.veBalance,
      _boostCheckpoint.veSupply,
      _boostCheckpoint.historyIndex,
      _balance.updateAt
    );
    if (uint256(_balance.updateAt) == block.timestamp) {
      _boostCheckpoint.veBalance = uint112(_currentVeBalance);
      _boostCheckpoint.veSupply = uint112(_currentVeSupply);
      _boostCheckpoint.historyIndex = uint32(_nextIndex);
      return (_currentRatio, _boostCheckpoint);
    }

    int256 _deltaVeBalance = int256(_currentVeBalance) -
int256(uint256(_boostCheckpoint.veBalance));
    int256 _deltaVeSupply = int256(_currentVeSupply) -
int256(uint256(_boostCheckpoint.veSupply));
    int256 _duration = int256(block.timestamp - _balance.updateAt);

    uint256 _prevTs = _balance.updateAt;
    uint256 _accumulatedRatio;
    // compute the time weighted boost from _balance.updateAt to now.
    uint256 _nowTs = ((_prevTs + WEEK - 1) / WEEK) * WEEK;
    for (uint256 i = 0; i < 256; ++i) {
      // it is more than 4 years, should be enough
      if (_nowTs > block.timestamp) _nowTs = block.timestamp;
      _accumulatedRatio += _currentRatio * (_nowTs - _prevTs);
      if (_nowTs == block.timestamp) break;
      uint256 _veBalance;
      uint256 _veSupply;
      {
        int256 dt = int256(_nowTs - _balance.updateAt);
        _veBalance = uint256((_deltaVeBalance * dt) / _duration +
int256(uint256(_boostCheckpoint.veBalance)));
        _veSupply = uint256((_deltaVeSupply * dt) / _duration +
int256(uint256(_boostCheckpoint.veSupply)));
      }
      (_currentRatio, _nextIndex) = _boostRatioAt(_balance, _veBalance,
_veSupply, _nextIndex, _nowTs);
      _prevTs = _nowTs;
      _nowTs += WEEK;
    }

    _boostCheckpoint.veBalance = uint112(_currentVeBalance);
    _boostCheckpoint.veSupply = uint112(_currentVeSupply);
    _boostCheckpoint.historyIndex = uint32(_nextIndex);
    return (_accumulatedRatio / uint256(_duration), _boostCheckpoint);
  }
```

Status

The development team has adopted our suggestions and removed the redundant code in commit 097ead8.

4.3.9 A malicious user could falsely increase the stored funds in the contract to boost the overall yield for all users.

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

Location

BoostableRebalancePool.sol#642

Description

In the calculation of the parameter `_boostedBalance`, the `_supply.amount` parameter is used, which comes from the historical snapshots of the total amount of fToken funds deposited by users under the contract each day. When this value increases, the boost ratio for the corresponding user also increases.

Consider a scenario: Towards the end of each day, a malicious user deposits a substantial amount of fTokens. At the beginning of a new day, the same user withdraws these fTokens. This behavior leads to a false increase in the total fund amount in the historical snapshots for each day. Consequently, it artificially raises the FXN token yield for all users under the contract. It's important to verify whether this behavior aligns with the intended design of the protocol.

```
function _boostRatioAt(
    TokenBalance memory _balance,
    uint256 _veBalance,
    uint256 _veSupply,
    uint256 startIndex,
    uint256 t
  ) internal view returns (uint256, uint256) {
    // @note since totalSupplyHistory is bounded by the number of days,
    // it should be ok to use while loop.
    unchecked {
      uint256 endIndex = numTotalSupplyHistory;
      while (startIndex < endIndex) {
        if (totalSupplyHistory[startIndex].updateAt > t) break;
        startIndex += 1;
      }
      if (startIndex > 0) startIndex -= 1;
    }
    TokenBalance memory _supply = totalSupplyHistory[startIndex];
    uint256 _realBalance = _getCompoundedBalance(_balance.amount,
_balance.product, _supply.product);
    if (_realBalance == 0) {
      return ((PRECISION * 4) / 10, startIndex);
    }
    uint256 _boostedBalance = (_realBalance * 4) / 10;
    if (_veSupply > 0) {
      // @audit the `_supply.amount` here refers solely to the total
amount of fTokens under the contract when the snapshot is taken.
      _boostedBalance += (((_veBalance * uint256(_supply.amount)) /
_veSupply) * 6) / 10;
    }
    if (_boostedBalance > _realBalance) {
      _boostedBalance = _realBalance;
    }

    return ((_boostedBalance * PRECISION) / _realBalance, startIndex);
  }
```

Status

The development team has confirmed that this design choice was made to align with the logic of the Curve protocol. It is acceptable that malicious user behavior will not cause a protocol imbalance.

5. Conclusion

After auditing and analyzing the dynamic allocation mechanism of FXN tokens in the f(x) protocol, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above.

Disclaimer

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